FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Implementation of an Android Framework for USB storage access without root rights

Magnus Jahnen

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Implementation of an Android Framework for USB storage access without root rights

Implementierung eines Android Frameworks für den Zugriff auf USB Speicher ohne Rootrechte

Author:	Magnus Jahnen
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Nils Kannengießer, M.Sc.
Date:	April 15, 2014

I assure the single handed composition of this bachelor thesis only supported by declared resources.

Munich, 15th of April, 2014 Magnus Jahnen

Acknowledgments

Many acknowledgment goes to the operating system chair of the TUM, especially to Prof. Dr. Uwe Baumgarten and Nils Kannengießer for the ability to write this thesis and especially to Nils for guiding me. The operating system chair also borrowed me a lot of Android devices to test my work!

I also want to thank Jan Axelson for his book USB Mass Storage[2]. Without this book the thesis would have been a lot more cumbersome. It was an important resource throughout the whole work.

All people who read, corrected and gave me hints for improvement on this thesis in advance also deserve credit. I want to thank all of them at this place!

Abstract

This bachelor thesis describes the implementation of an Android framework to access mass storage devices over the USB interface of a smartphone. First the basics of USB (i.e. interfaces, endpoints and USB On the go) and accessing USB devices via the official Android API are discussed. Next the USB mass storage class is explained, which was designed by the USB-IF to access mobile mass storage like USB pen drives or external HDDs. For communication with mass storage devices, most important are the bulk-only transfer and the SCSI transparent command set. Furthermore file systems, for accessing directories and files, are described. This thesis focuses on the FAT32 file system from Microsoft, because it is the most commonly used file system on such devices.

After the theory part it is time to look at the implementation of the framework. In this section, the first concern is the purpose in general. Then the architecture of the framework and the actual implementation are presented. Important parts are discussed in detail.

The thesis finishes with an overview of the test results on various Android devices, a short conclusion and an outlook to future developments. Moreover the current status of the developed framework is visualized.

Contents

Outline of the Thesis

Part I: Introduction and Theory

Chapter 1: Introduction

Overview of the thesis and its purpose.

Chapter 2: Basics about USB

Basics of USB, how it is structured and how it generally works.

Chapter 3: USB on Android

Introduction to the USB API on Android. How can USB devices be enumerated and accessed, and how can the communication be initiated?

Chapter 4: USB Mass storage class

Overview of the USB mass storage class. The main goal is to understand SCSI commands to communicate with the device and read and write from and to its storage.

Chapter 5: File systems

File systems in general and a detailed description of how FAT32 from Microsoft works.

Part II: Implementation

Chapter 6: Purpose and Overview

Purpose and requirements of the implementation and a short overview of the framework's architecture.

Chapter 7: Inside the packages

Deeper insight into the developed framework, it's packages, classes and interaction between the important parts.

Part III: Quality Management

Chapter 8: Testing

Results of different tests regarding the developed framework on various devices.

Part IV: Results

Chapter 9: Summary
Short summary of the current status of the framework and a conclusion of the thesis.

Chapter 10: Outlook
Some hints about further development and advices what future may hold.

Acronyms

adb Android Debug Bridge

API Application Programming Interface

app Application

ASCII American Standard Code for Information Interchange

ATAPI Advanced Technology Attachment with Packet Interface

BBB bulk-only transport

BIOS Basic input/output system

btrfs B-tree file system

CBI Control/Bulk/Interrupt

CBS Command Status Wrapper

CBW Command Block Wrapper

CHS Cylinder-head-sector

exFAT Extended File Allocation Table

ext3 Third extended file system

ext4 Fourth extended file system

FAT File Allocation Table

FS File System

GPT GUID Partition Table

GUID Globally Unique Identifier

HDD Hard Disk Drive

HFS+ Hierarchical File System Plus

ID Identifier

IP Internet Protocol

LBA Logical Block Address

LFN Long File Name

LUN Logical Unit Number

MBR Master Boot Record

NTFS New Technology File System

OS Operating System

RBC Reduced Block Commands

ROM Read-only-memory

SCSI Small Computer System Interface

SD Secure Digital

SPC SCSI Primary Commands

UEFI Unified Extensible Firmware Interface

UML Unified Modeling Language

USB OTG USB On the go

USB-IF USB Implementers Forum

USB Universal Serial Bus

xml Extensible markup language

Part I.

Introduction and Theory

1. Introduction

Since Android 3.1, which was originally designed for tablet computers, a lot of Android devices come with USB host support (USB On the go). That means a normal Android tablet or phone can not only act as an USB client when connected to a computer. It can also act as an USB host for peripherals by powering the bus with the needed five Volt and changing into USB host mode which enables enumeration of connected USB devices[8]. Android currently supports interrupt, bulk and control transfers[1]. That means almost every USB device can, theoretically, be used with an Android device[2]. The Android host API allows to communicate with connected USB devices, i.e. a high level USB driver can be written in Java.

Whereby the idea of connecting a USB mass storage device like USB flash drives or external HDDs is not far-fetched. Especially when looking at recent occurrences where a lot of devices lack a slot for SD-Cards and only offer a solid, mostly small, internal storage. Unfortunately the stock Android comes without support for USB storage devices. That means when connecting mass storage devices to an Android phone, nothing happens. The data cannot be accessed via a file manager or something similar. On rooted devices this is possible because the alternative Android ROMs often provide support for it. But with the Android USB Host API it should also be possible to access such devices without rooting the device and flashing an alternative ROM. The only thing needed is to implement the low level USB communication via e.g. bulk transfers and the abstraction of directories and files via a file system.

Currently there are two applications in the Google Play Store which allow accessing mass storage devices without root rights! First there is a plugin for the Total Commander called *USB-Stick Plugin-TC*. The plugin extends the Total Commander application by USB mass storage access. It currently supports FAT12, FAT16, FAT32, exFAT and NTFS (read only). There is a free trial version available. The second application is called *Nexus Media Importer*. It supports FAT16, FAT32 and NTFS (also read only). There is no free trial version available. In general both apps support USB sticks, external HDDs and card readers.

The disadvantage of both applications is that there is no solution to access the mass storage from other apps. That means all accessed data has to be cached and copied to the internal storage before any other app can use it. Unfortunately it seems that these limitations cannot be removed.

Both applications are proprietary and the source code is not available for reference or modification. This is why an open source Android framework for accessing mass storage devices is developed in this bachelor thesis. The desired license is the very liberal Apache License, Version 2.0, the same license Android is licensed under.

Due to the same licensing it would be possible for Google to integrate this solution into the official Android. But there are some factors which make the integration unlikely. First

[1] Isochronous transfers are currently unsupported[9].

[2] Webcams or audio devices mostly use isochronous transfers and can thus not be used at the moment.

of all, necessary things, like file systems (e.g. FAT32) or the SCSI transparent command set, for mounting USB mass storage are already implemented in the underlying Linux kernel. Google just deactivated the support for it. Second, with the solution described in this thesis, only apps which use the framework can access USB storage devices. It would be more straightforward if the connected devices would be mounted in the normal unix file system like SD-cards. For example under */mnt/usbstick0*. This would allow other apps to easily access data from USB mass storage without extra changes to the application. Therefore it is very unlikely that Google will integrate this framework into the official Android. If Google decides to support mounting USB mass storage devices, they will most likely enable support for it in the kernel and mount the devices in the normal file system, like some manufacturers (e.g. Samsung) already do.

Another reason for Google not to implement the support for mass storage devices over USB, is to move more people to use their own cloud service Google Drive. But maybe, if Google notices the growing popularity of applications allowing access of USB mass storage, Google may enable support for it in their mobile operating system.

Numbers in this thesis

There are a lot of numbers in this thesis. If the number has a trailing 'h', for example 08h, this number shall be interpreted as a hex number. Numbers without this 'h' shall be interpreted as decimal numbers.

2. Basics about USB

USB stands for Universal Serial Bus and is a standard for a serial bus system, for connecting multiple peripheral devices to a personal computer. The first version was introduced by Intel in 1996. Today the specification is done by the USB Implementers Forum (USB-IF). The USB-IF is a corporation founded by various companies which work non-profit on the USB specification. In USB communication there are two kinds of devices, one USB host controller (e.g. computer) and one or more clients (slaves). The host is responsible for the communication process. The client only sends data when the host asks for it. The USB host is responsible for powering the connected client, thus an external power source is only necessary in some special cases[20].

2.1. Client device hierarchy

A USB client is structured into four different USB descriptors:

- Device descriptor
- Configuration descriptor
- Interface descriptor
- Endpoint descriptor

The device descriptor represents the USB device as a whole device which is connected to the USB bus. This can for example be a loud speaker with volume control buttons.

The configuration descriptor represents the current state of the USB device. This can for example be standby or active.

A USB interface descriptor describes every logical device which belongs to the USB device. Often USB devices consist of multiple logical device units. For example a loud speaker could consist of the speakers as an audio device and buttons to control the volume as a human interface device.

Lastly there are endpoint descriptors which represent unidirectional communication pipes. This is where the actual communication happens. Endpoints can either be of type IN (device to host) or OUT (host to device). Additionally there are four different types of endpoints, to fit the different requirements of communication[11].

2.2. Endpoints

Every USB device has different requirements on the underlying communication pipe. To satisfy the different needs the USB protocol offers four different types of communication (endpoints).

Control endpoints are used to configure the device and retrieve status information. Control transfers are typically very small. Every device has a control endpoint called endpoint 0 which plays an important role at insertion time[11].

Interrupt transfers carry a small amount of data to the host every time the host asks for it. This happens at a fixed rate resulting in a fixed and guaranteed bandwidth. These transfers are used by human interface devices (e.g. mouse, keyboard, gamepads) which need a low latency and a low packet size.

Next there are bulk endpoints. These are useful when the amount of data transferred varies often and happens infrequently. The remaining available bandwidth the bus offers is used. Hence, there is no guarantee on bandwidth or latency. However bulk transfers offer consistent data transfers, meaning that no data is lost. They are typically used for printers, network or mass storage devices. Everywhere data loss is unacceptable and no guaranteed bandwidth is needed.

Finally there are the isochronous transfers. They offer a guaranteed bandwidth while resigning consistency. The guaranteed bandwidth is mostly as fast as possible and valuable for real time transfers (e.g. audio or video). Mostly these transfers are used for webcams or audio devices (e.g. external audio cards/audio interfaces)[11].

2.3. USB On the go

As already mentioned, in USB communication there is always a host (master) and a client (device). The host initiates the communication and acts as a master. This means that the client can only send data after being explicitly asked to do so by the host. The client is only able to signal that it requires attention. Then the host must react and ask for receiving data. When connecting a smartphone or tablet to the computer, the computer acts as the host and the smartphone acts as the client device. That means the smartphone normally acts as a client device and not as the USB host. In the desired constellation described in this thesis, however, it has obviously to act as a host.

For that reason the USB-IF developed the USB On the go (USB OTG) feature in 2001 as part of the USB 2.0 specification[20]. This feature allows a USB device to act as a client or either as a host depending on the present situation. To use the USB OTG mode, on a smartphone, a special USB OTG adapter is needed. This is necessary for two reasons. First for signaling that the smartphone should act as a host and not as usual as a client and second because most smartphones and tablets do not provide a normal USB port of type A. Instead they offer a mini (older devices) or micro port of type Mini-A or Micro-B[21].

3. USB on Android

As already mentioned, Google added USB features to the Android OS with Android 3.1 Honeycomb. There are two different modes Android keeps under control. The already mentioned host support and a special USB accessory mode. The accessory mode is only available on Android. It is supposed for developing USB host hardware specifically designed for Android devices, where the accessory is the USB host and powers the Android device[7]. The Android device is the client and can for example charge through and interact with the hardware (e.g. playing music through external speakers).

The USB Accessory mode is also backported to Android 2.3.4[7]. The developed framework solely relies on the USB host functionality since a memory stick is a USB client.

3.1. USB Host API

Android offers classes to enumerate, connect to and communicate with connected USB devices. Table 3.1 gives an overview of the classes which can be found in the package *android.hardware.usb*.

Table 3.1.: USB host APIs, compare to [8]

Class	**Description**
UsbManager	Allows the enumeration and communication with connected USB devices.
UsbDevice	Represents a connected USB device and contains methods to access its identifying information, interfaces, and endpoints.
UsbInterface	Represents an interface of a USB device, which defines a set of functionality for the device. A device can have one or more interfaces.
UsbEndpoint	Represents an interface endpoint, which is a communication channel for this interface. An interface can have one or more endpoints, and usually has input and output endpoints for two-way communication with the device.
UsbDeviceConnection	Represents a connection to the device, which transfers data on endpoints. This class allows sending data back and forth synchronously or asynchronously.
UsbRequest	Represents an asynchronous request to communicate with a device through a UsbDeviceConnection.
UsbConstants	Defines USB constants that correspond to definitions in *linux/usb/ch9.h* of the Linux kernel.

The UsbRequest class is only needed when communicating asynchronously[1]. The rough procedure of getting in touch with a USB device includes following steps:

1. Retrieve the desired UsbDevice via the UsbManager
2. Get the appropriate UsbInterface and UsbEndpoint
3. Begin the communication by opening a UsbDeviceConnection via the UsbEndpoint

To understand the following sections, fundamental knowledge of Android programming is recommended. Basics[2] are not described in detail here. An introduction to Android programming can be found in the official Android developer documentation[3].

3.2. Enumerating devices

To enumerate through all connected USB devices the singleton UsbManager is used. It allows looping through the device list. The device list is returned by the method getDeviceList() of the UsbManager.

Listing 3.1: Enumerating connected USB devices

```
UsbManager usbManager = (UsbManager)
   context.getSystemService(Context.USB_SERVICE);

for(UsbDevice device : usbManager.getDeviceList().values()) {
      Log.i(TAG, "found usb device: " + device);
}
```

Accessing the UsbInterface and the UsbEndpoint is also straightforward. The UsbDevice has a method to get the desired interfaces and the UsbInterface has a method to get the desired endpoints on the other hand. Listing 3.2 illustrates the process.

Listing 3.2: Accessing UsbInterface and UsbEndpoint

```
UsbManager usbManager = (UsbManager)
   context.getSystemService(Context.USB_SERVICE);

for(UsbDevice device : usbManager.getDeviceList().values()) {
      Log.i(TAG, "found usb device: " + device);

      int interfaceCount = device.getInterfaceCount();
      for(int i = 0; i < interfaceCount; i++) {
            UsbInterface usbInterface = device.getInterface(i);
            Log.i(TAG, "found usb interface: " + usbInterface);

            int endpointCount = usbInterface.getEndpointCount();
```

[1] Asynchronous communication is passed in the implementation of the framework.

[2] Following things, for example, are seen as basic: Activity, Intent, PendingIntent, IntentFiler, Broadcast, BroadcastReceiver.

[3] `http://developer.android.com`

```
            for(int j = 0; j < endpointCount; j++) {
                  UsbEndpoint endpoint = usbInterface.getEndpoint(j);
                  Log.i(TAG, "found usb endpoint: " + endpoint);
            }
      }
}
```

While looping through the devices, it can easily be checked if any device fits the desired needs. UsbDevice and UsbInterface offer methods to get the class, subclass and the protocol of the device, resp. the interface. The UsbEndpoint class has methods to get the type, the direction and other attributes of the corresponding endpoint. The UsbDevice also offers methods to get the vendor and product ID of the connected device.

3.3. Requesting permission for communication

After discovering a suitable USB device Android requires the user to accept the communication between an application and the USB device first. To do so the permission to communicate with the USB device has to explicitly be requested. A dialog is shown to the user asking for permission, where the user can click okay or cancel. Therefore the UsbManager offers a method called requestPermission which takes a UsbDevice and a PendingIntent as parameter. The PendingIntent in this case is a Broadcast which can be received via registering a BroadcastReceiver with a specific IntentFilter.

Listing 3.3 shows how a BroadcastReceiver, for receiving notifications about the permission, can look like. First the intent action is validated, this step is only needed if the BroadcastReceiver receives multiple different actions. In this example this is not the case. After that the UsbDevice can be accessed via an extra of the intent. Another extra of the intent is the permission state. If it is granted the extra is true and it is permitted to communicate with the device.

Listing 3.3: Permission BroadcastReceiver [8]

```
private static final String ACTION_USB_PERMISSION =
   "com.android.example.USB_PERMISSION";
private final BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {

   public void onReceive(Context context, Intent intent) {
      String action = intent.getAction();
      if(ACTION_USB_PERMISSION.equals(action)) {
         synchronized (this) {
            UsbDevice device =
                (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);

            if(intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED,
                false)) {
               if(device != null){
                 //call method to set up device communication
               }
            }
            else {
                Log.d(TAG, "permission denied for device " + device);
            }
         }
      }
   }
};
```

The next step is to register this BroadcastReceiver that it can actually receive broadcasts from the system. This normally happens in the onCreate method of an Activity via the method registerReceiver which takes the BroadcastReceiver (mUsbReceiver) and the IntentFilter as parameter. The IntentFilter uses the ACTION_USB_PERMISSION string, declared in Listing 3.3, to filter undesired actions.

Listing 3.4: Registering the BroadcastReceiver [8]

```
IntentFilter filter = new IntentFilter(ACTION_USB_PERMISSION);
registerReceiver(mUsbReceiver, filter);
```

The last step consists of requesting the permission using the UsbManager:

Listing 3.5: Requesting permission [8]

```
UsbDevice device;
...
mPermissionIntent = PendingIntent.getBroadcast(this, 0, new
   Intent(ACTION_USB_PERMISSION), 0);
mUsbManager.requestPermission(device, mPermissionIntent);
```

3.4. Communication

After succeeding all necessary steps to set up the device, communication is possible. To do so the desired device has to be opened and an UsbDeviceConnection has to be retrieved. The class UsbDeviceConnection offers a method to claim a certain UsbInterface. After that communication is possible via the UsbDeviceConnection. It offers methods like bulkTransfer or controlTransfer.

Listing 3.6: Communicating with a connected device [8]

```
private Byte[] bytes
private static int TIMEOUT = 0;
private boolean forceClaim = true;
...
UsbInterface intf = device.getInterface(0);
UsbEndpoint endpoint = intf.getEndpoint(0);
UsbDeviceConnection connection = mUsbManager.openDevice(device);
connection.claimInterface(intf, forceClaim);
connection.bulkTransfer(endpoint, bytes, bytes.length, TIMEOUT); //do
   in another thread
```

Listing 3.6 uses, for simplicity reasons, the first interface and endpoint. Normally the endpoint to communicate with, should be chosen wisely by examining for example the interface class, or the vendor ID of the device. A control transfer would look similar.

3.5. Tearing down the communication

When the communication between the Android application and the USB device is done, it has to be shut down. This is done by releasing the interface and closing the connection. Listing 3.7 gives an example about how to do that.

Listing 3.7: Closing communication

```
public void close() {
      Log.d(TAG, "close device");
      boolean release = deviceConnection.releaseInterface(usbInterface);
      if(!release) {
            Log.e(TAG, "could not release interface!");
      }
      deviceConnection.close();
}
```

3.6. Listening to attach and detach events

Android does not only allow enumerating connected devices, an application can also register for attach and detach events of USB devices. The application then gets notified whenever a USB device is connected to, or disconnected from the Android device. There are two different ways to do that. The first one is via a BroadcastReceiver, the second one via the AndroidManifest.xml file. The second method has the advantage that the application is notified even if it has not been started before.

3.6.1. Via BroadcastReceiver

Listing 3.8: Attach and detach notification of USB devices via a BroadcastReceiver

```
BroadcastReceiver mUsbReceiver = new BroadcastReceiver() {
   public void onReceive(Context context, Intent intent) {
      String action = intent.getAction();

    if (UsbManager.ACTION_USB_DEVICE_ATTACHED.equals(action)) {
         UsbDevice device =
            (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
         if (device != null) {
            // call method that sets up and initiates communication
               with the device
         }
      }
```

```
        if (UsbManager.ACTION_USB_DEVICE_DETACHED.equals(action)) {
            UsbDevice device =
                (UsbDevice)intent.getParcelableExtra(UsbManager.EXTRA_DEVICE);
            if (device != null) {
                // call method that cleans up and closes communication with
                    the device
            }
        }
    }
};
```

To use this BroadcastReceiver it has to be registered in an Activity or Service with the corresponding IntentFilter like this:

Listing 3.9: Registering the BroadcastReceiver with the desired actions

```
IntentFilter filter = new IntentFilter();
filter.addAction(UsbManager.ACTION_USB_DEVICE_ATTACHED);
filter.addAction(UsbManager.ACTION_USB_DEVICE_DETACHED);
registerReceiver(mUsbReceiver, filter);
```

3.6.2. Via AndroidManifest.xml

If an application wants to be notified about the attachment of an USB device this can also be specified in the AndroidManifest.xml. This has the advantage that the application does not have to be started before. In fact it is started when a desired USB device is connected. The user is then asked if he wants to start the application which can handle the attached device. The next benefit is that the step of requesting permission, described in 3.3, is not required because the user already gave his consent by allowing the application to start.

Additionally a device filter can be specified, which allows the application to be notified only if an appropriate device is attached. Following attributes can be specified[8]:

- Vendor ID
- Product ID
- Class
- Subclass
- Protocol (device or interface)

Below is an example how a device filter could look like:

Listing 3.10: Example device filter [8]

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
   <usb-device vendor-id="1234" product-id="5678" class="255"
       subclass="66" protocol="1" />
</resources>
```

This resource file should be located at *res/xml/device_filter.xml* in the project directory[8]. The device filter can then be used in the AndroidManifest.xml, like in Listing 3.11.

Listing 3.11: AndroidManifest.xml [8]

```
<manifest ...>
   <uses-feature android:name="android.hardware.usb.host" />
   <uses-sdk android:minSdkVersion="12" />
   ...
   <application>
      <activity ...>
         ...
         <intent-filter>
            <action
               android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
               />
         </intent-filter>

         <meta-data
            android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
            android:resource="@xml/device_filter" />
      </activity>
   </application>
</manifest>
```

The intent filter for the action uses, again, the USB_DEVICE_ATTACHED string, like when using a BroadcastReceiver. This time no broadcast is sent, but an activity is started. The manifest also contains a *uses-feature* entry, because not all Android devices guarantee to support the USB host feature[8]. The minimum sdk version is set to 12 here, because on lower API levels the USB host API is not available.

After that the UsbDevice can be accessed anywhere within the Activity like this:

Listing 3.12: Accessing the UsbDevice in the Activity, compare to: [8]

```
UsbDevice device = (UsbDevice)
   getIntent().getParcelableExtra(UsbManager.EXTRA_DEVICE);
```

4. USB Mass storage class

Most USB devices are of same type. To reduce development effort and allow OS designers offering generic drivers for a great range of different devices, a lot of device types are standardized. These different types are called classes in USB. There are for example standardizations for printers, USB hubs, audio or video devices, human interface devices and mass storage devices[15]. The focus in the following, is on the mass storage class.

Every mass storage device has at least one interface descriptor with the class code 08h, which stands for the mass storage class. The mass storage class is not defined in the device descriptor! The USB interface has exactly two endpoint descriptors. One IN endpoint to read from the device and one OUT endpoint to write to the device[2]. Reading and writing in this case does not necessarily mean reading or writing on the actual storage medium, this is described later.

There are two different types regarding the mass storage class. There is the bulk-only transport (BBB) mechanism which is the most common one. All newer devices follow that standard. Then there is the Control/Bulk/Interrupt (CBI) standard which is no longer important, because the USB-IF recommends using the BBB approach[2].

4.1. Bulk-only Transport

Unlike the name suggests there are two control requests in the BBB specification. The first one is a reset request to prepare the device for the next command. The second is used to get the maximum LUN (Get Max LUN request). This request informs about the number of standalone logical units the mass storage device supports[2].

As mentioned, the interface class has to be set to 08h for the mass storage class. The subclass of the interface descriptor can have different values and specifies the supported protocols used to read and write data from and to the mass storage. Table 4.1 gives an overview of the different protocols.

Table 4.1.: Overview subclass protocols [2]

01h	Reduced Block Commands (RBC)
02h	SFF-8020i, MMC-2 (ATAPI) (CD/DVD drives)
03h	QIC-157 (tape drives)
04h	USB Floppy Interface (UFI)
05h	FF-8070i (ATAPI removable rewritable media devices)
06h	SCSI transparent command set

For the purpose described in this thesis, the SCSI transparent command set is the most important one, which is explained in the following chapter. The RBC is not even implemented in Windows, but in the Linux kernel[2]. The other protocols refer to other types of storage media which are not covered by this thesis.

4.2. SCSI transparent command set

Every SCSI command the host sends to the client is enclosed by a so called Command Block Wrapper (CBW). Sending this CBW is always the first thing when host and device exchange data. After transmitting the CBW, raw data can be transferred. The direction of that data can either be from the host to the device or vice versa. In this document, from here on, this phase is called the data, transport or transfer phase. Some commands do not need the data phase. In the last step the client device sends a Command Status Wrapper (CSW) to the host to inform about any failures or success.

The CBW is always 31 bytes long including the enclosing SCSI command. The host sends it through the OUT endpoint to the device. Following table illustrates the CBW:

Table 4.2.: Command Block Wrapper, compare to: [2]

Field Name	Bits	Description
dCBWSignature	32	Fixed value of 43425355h to identify the CBW.
dCBWTag	32	Corresponds to dCSWTag in CSW.
dCBWDataTransferLength	32	The number of bytes which will be sent by the host in the transfer phase or the number of bytes the host expects to receive in the transfer phase. Depends on bmCBWFlags.
bmCBWFlags	8	If bit 7 is set to 0 the data transfer is from host to device, if it is set to 1 from device to host. All other bits are unused. If there is no transfer phase this value shall be zero.
Reserved	4	-
bCBWLUN	4	The LUN the command is directed to.
Reserved	3	-
bCBWCBLength	5	The length of the actual SCSI command located in the CBWCB field.
CBWCB	128	The SCSI command the client shall execute.

The dCBWTag is useful to associate the CSW with the CBW. The device uses the same value stored in the dCBWTag in the dCSWTag of the CSW. If multiple CBWs are sent at the same time the corresponding CSWs can easily be found with the help of the tag.

What data is transferred and how, is discussed in the sections about the different SCSI commands. For now it is ignored and the CSW is introduced in table 4.3. The CSW is always 13 bytes.

Table 4.3.: Command Status Wrapper [2]

Field Name	Bits	Description
dCSWSignature	32	Fixed value of 53425355h to identify the CSW.
dCSWTag	32	Value of dCBWTag from the CBW the device received.
dCSWDataResidue	32	This indicates the number of bytes in the transport phase the device has not yet processed. Should be 0 if all data has been processed.
bCSWStatus	8	0 if command successfully passed, 1 if there was an error and 2 on a phase error.

The dCSWDataResidue holds the difference between the dCBWDataTransferLength and the number of bytes the device either processed when the host sends data or the number of bytes the device already sent to the host. In most cases all data can be transferred in one transfer phase meaning dCSWDataResidue is mostly zero.

The bCSWStatus informs about the success of executing the desired SCSI command. A value of zero indicates success. If this field is set to one there was an error executing the command. The host should then issue a SCSI REQUEST SENSE command to get more information about what went wrong[2]. More on this SCSI command later. If this value is two the host should perform a reset recovery. The reset consists of a bulk-only mass storage reset, which is one of the class specific commands and a Clear Feature HALT on the IN and OUT endpoint[2, 6].

The fields in the CBW and CSW are all serialized in little endian style.

4.3. SCSI commands

The Small Computer System Interface (SCSI) is a standard for communicating between computers and peripheral devices. It is most commonly used for hard disks and other storage devices, but it can also be used for example for scanners[19]. SCSI commands are used to get general information about the connected storage device, but also for reading and writing data from and to the device's storage. The USB mass storage class also uses this well established standard.

There are a lot of different SCSI commands and not every device supports every command. To determine which commands are supported by a specific device, the host issues a SCSI INQUIRY command. Every device has to support this command and deliver a meaningful response to it. The device discloses, with the information included in the INQUIRY response, which commands are supported, i.e. which standardization it follows. In practice the most commonly supported commands are[2]:

- INQUIRY
- READ CAPACITY(10)
- READ(10)

- REQUEST SENSE
- TEST UNIT READY
- WRITE(10)

Every device should support at least these commands! Every SCSI command starts with the operation code, also called OPCODE (one byte), which identifies the command. The following data depends on the specific command. The ten after some commands describes the length of the command in bytes. There are for example different READ commands, READ(6), READ(10), READ(12), READ(16) and READ(32)[12]. These commands all differ in their length. This is needed because in the READ(6) command, the logical block address field which is used to address a block is only 16 bit. However devices with a big storage cannot use this command, because the whole storage cannot be addressed with a 16 byte value. Thus in the READ(10) command, which is the most commonly used read command, the address field is 32 bit.

SCSI commands use the big endian style for storing fields bigger than one byte.

4.3.1. INQUIRY

As already mentioned the INQUIRY command is used to get general information about the connected storage device. A host should issue this command to determine the supported SCSI commands by the device. The response to the INQUIRY command is transferred in the transport phase between sending the CBW which includes the INQUIRY command and receiving the CSW. The direction of the transport phase is from the client to the host.

Table 4.4.: INQUIRY command, compare to: [12]

Byte	Description
0	Operation code (12h)
1	Bit 0: EVPD, Bit 1: Obsolete, Bit 2-7: Reserved
2	Page Code
3-4	Allocation Length (Byte 3: MSB, Byte 4: LSB)
5	Control

The most important fields in the INQUIRY command are the operation code and the allocation length. The allocation length tells the storage device how many bytes the host has allocated for the INQUIRY response. The device then replies with an answer not larger than the allocation length. The Allocation Length field should be at least five (bytes). The EVPD[1] and the page code are used to get more information about the vital product data. If the EVPD bit is set to one, the device should return the part of the vital product data specified in the field page code. If the EVPD bit is set to zero only the standard INQUIRY data shall be returned[12]. This thesis describes only the latter case.

The Allocation Length field and the Control field are commonly used fields which occur in various SCSI commands[12].

[1] Enable Vital Product Data

The response to the standard INQUIRY request should contain at least 36 bytes[12]. Nevertheless, it is up to the manufacturer of the device, how big the response is, because the response can include vendor specific information[12]. Bytes 5 to N consist of fields not discussed here, because they are less important or vendor specific information.

Table 4.5.: Standard INQUIRY data, compare to: [2, 12]

Byte	Description
0	Bit 0-4: Peripheral device type, Bit 5-7: Peripheral Qualifier
1	Bit 0-6: Reserved, Bit 7: RMB
2	Version
3	Bit 0-3: Response data format, Bit 4: HISUP, Bit 5: NORMACA Bit 6,7: Obsolete
4	Additional length (N-4)
5-N	...

The peripheral device type shall always be zero. This indicates that a peripheral device is connected to the logical unit. The peripheral qualifier describes the connected device. If this field is set to zero the connected device is a direct access block device. A value of two means a printer device and a value of five indicates a CD or DVD drive[2, 12]. This value shall also always be zero for a direct access block device, because the direct acess block device is the only type of device the framework shall support.

The RMB bit indicates if the device is removable or not. Zero indicates a non removable device and one a removable device. USB flash drives are removable devices, but they have a fixed media unlike card readers. But Microsoft suggests that flash drives declare they have removable media, and thus some flash drives do this[2].

The Version field indicates which standard of the SPC (SCSI Primary Commands) the device follows. If the value is zero the device does not comply to any standard. If the value is three or four, the device complies to the SPC or SPC-2 standard[2, 12].

The Response Data Format field must equal to two, because values lower than two are obsolete and values bigger than two are reserved[12].

The additional length provides information about how many bytes are remaining in the response. The additional data is not important at the moment.

4.3.2. TEST UNIT READY

This command tests if the storage device is ready to use. It does not have a transport phase. If the device is ready to use the CSW status is set to successful and if not to a status, indicating failure. In the latter case the host should issue a SCSI REQUEST SENSE, to get information about what went wrong. When the device has removable media, this command can be used to check if a media is currently present[2].

Table 4.6.: TEST UNIT READY command [12]

Byte	Description
0	Operation Code (00h)
1-4	Reserved
5	Control

4.3.3. READ CAPACITY

The READ CAPACITY command is used to determine the storage space of a device. The device tells the host the logical block address (LBA) of the last block and the size in bytes of a single block. The total number of blocks is the LBA of the last block plus one. The direction of the transport phase is from the peripheral to the computer.

Table 4.7.: READ CAPACITY(10) command [12]

Byte	Description
0	Operation Code (25h)
1	Bit 0: Obsolete, Bit 1-7: Reserved
2-5	Logical Block Address (Byte 2: MSB, Byte 5: LSB)
6,7	Reserved
8	Bit 0: PMI, Bit 1-7: Reserved
9	Control

If the PMI (partial media indicator) bit is set to zero, the logical block address must also be set to zero. The device then returns information of the last logical block. If the PMI bit is set to one, the Seagate manual on SCSI commands says: "A PMI bit set to one specifies that the device server return information on the last logical block after that specified in the LOGICAL BLOCK ADDRESS field before a substantial vendor specific delay in data transfer may be encountered."[12]

The response transferred in the transport phase looks like this:

Table 4.8.: READ CAPACITY(10) response, compare to [2, 12]

Byte	Description
0-3	Last Logical Block Address (Byte 0: MSB, Byte 3: LSB)
4-7	Block length in bytes (Byte 4: MSB, Byte 7: LSB)

4.3.4. READ(10) and WRITE(10)

The READ(10) command requests the device to read the specified blocks from the storage and to transfer them to the host. The logical block address included in the command specifies the block where reading shall begin. The Transfer Length field holds the amount of contiguous blocks that shall be read. The device then transmits the requested data in the data transport phase. The device does not have to care about the actual data, it transfers the data to the host, just like it is saved on the storage. Table 4.9 shows how the READ(10) command is constructed.

Table 4.9.: READ(10) command [12]

Byte	Description
0	Operation Code (28h)
1	Bit 0: Obsolete, Bit 1: FUA_NV, Bit 2: Reserved, Bit 3: FUA, Bit 4: DPO, Bit 5-7: RDPROTECT
2-5	Logical Block Address (LBA) (Byte 2: MSB, Byte 5: LSB)
6	Bit 0-4: Group Number, Bit 5-7: Reserved
7,8	Transfer Length (Byte 7: MSB, Byte 8: LSB)
9	Control

For this thesis only the LBA and Transfer Length fields are important. The other fields shall remain zero. They are responsible, for example, to specify caching behavior or read protection[2].

The WRITE(10) command is formatted exactly like the READ(10) command except that the operation code is 2Ah and the RDPROTECT field is called WDPROTECT. The direction of the transport phase is, of course, the other way round, from computer to the device. The direction has to be specified correctly in the CBW!

Logical Block Address

Every mass storage device is structured in blocks. These blocks have a defined size. The size of each block can be determined by issuing a READ CAPACITY(10) command. The blocks are numbered consecutively beginning from zero to the amount of blocks minus one. This number is called logical block address, short LBA. With the LBA every block can easily be addressed and accessed. The READ(10) and WRITE(10) SCSI commands use this method of addressing for reading and writing data from and to the storage medium. The transfer length specifies how many blocks, including the block at the LBA, shall be transferred. That means reading or writing begins with a block defined through the LBA with as many directly consecutive blocks as desired.

4.3.5. REQUEST SENSE

If the device fails executing a SCSI command requested by the computer, an unsuccessful CSW status is set. The computer then knows that something went wrong, but it does not know what went wrong. To get more information about a specific error the host can issue a REQUEST SENSE command, to request the sense data from the device.

[2] Way back, pen drives with a physical switch for write protection, were pretty common.

Table 4.10.: REQUEST SENSE command [12]

Byte	Description
0	Operation Code (03h)
1	Bit 0: DESC, Bit 1-7: Reserved
2-3	Reserved
4	Allocation Length
5	Control

The DESC bit describes if the fixed sense data or the descriptor format sense data shall be transferred. A value of zero requests the fixed sense data[12].

The Allocation Length field indicates, like in the INQUIRY command, how many bytes the host has allocated for data to be received. The device does not send more data than the host has actually allocated. But that means that some information can be lost if the data requested is actually bigger than the allocated space.

Table 4.11 shows the contents of the fixed sense data transferred from the device to the computer in the data transport phase. The size of the sense data normally is 252 bytes, with vendor specific information beginning at byte 18.

Table 4.11.: Fixed SENSE data, compare to: [2, 12]

Byte	Description
0	Bit 0-6: Response Code, Bit 7: VALID
1	Obsolete
2	Bit 0-3: SENSE KEY, Bit 4: Reserved, Bit 5: ILI, Bit 6: EOM, Bit 7: FILEMARK
3-6	Information (Byte 3: MSB, Byte 6: LSB)
7	Additional sense length (N-7)
8-11	Command-specific information (Byte 8: MSB, Byte 11: LSB)
12	Additional Sense Code
13	Additional Sense Code Qualifier
14	Field replaceable unit code
15-17	Bit 0-20: Sense key Specific, Bit 21: SKSV
18-N	...

A detailed description of these fields can be found in the SCSI Commands Reference Manual from Seagate[12].

5. File systems

5.1. General

With the USB bulk transfers and the SCSI commands it is now possible to read and write raw data from and to the device. That means one can access the bytes stored on the medium. The only thing missing is some abstraction to handle the raw data. To do this on mass storage devices the most commonly used approaches are partitions and file systems. A file system is a way to organize data in directories and files with human readable names. Thus the user can easily find things without knowing something about the addressing methods of a mass storage device.

Directories

A directory is a container for files and other directories. With the help of a directory the contents of a file system or mass storage device can easily be structured in a tree based way. Every file system has a root directory which is the directory at the top of the file system.

Files

Unlike directories, files do not help structuring the contents, but hold the actual data the user wants to save and later access again.

5.1.1. Examples

Today there are many different file systems. Most commonly used file systems are FAT32 and NTFS from Windows background, ext3, ext4 and btrfs from the Linux/Unix background and HFS+, also called Mac OS Extended, developed by Apple for their OS X operating system. All listed file systems, except FAT32, use binary trees or something similar to structure the contents of the directories.

On USB mass storage devices the most popular file systems are FAT32 and NTFS, because obviously most people use the Windows operating system. The NTFS specification is not published by Microsoft. Thus it is very hard to support NTFS on other systems, nevertheless there is NTFS support in the Linux kernel. The FAT32 specification is publicly available and can be downloaded from the Microsoft website[1]. Because of the lack of support for unix like file systems on Windows, the Linux and OS X file systems are only used by users who do not need to exchange data with Windows machines. Therefore the FAT32 file system, which has an open specification and is the mostly used file system on SD-cards and pen drives, is described in the following sections in detail.

[1] http://msdn.microsoft.com/en-us/windows/hardware/gg463080.aspx

5.1.2. Partition table

Before having a closer look at the FAT32 file system, partition tables have to be explained. A physical drive can have multiple partitions which operate independently from each other. They can also have different file systems. In Windows for example every partition is handled by a separate drive letter.

A partition table holds the information needed for identifying the different partitions on the disk. This information includes where the partition on the disk starts, ends or how many blocks it occupies and with which file system the partition is formatted. There are two different partition tables most commonly used today. The Master Boot Record (MBR) and the GUID partition table (GPT) which is part of the UEFI standard[17]. The Master Boot Record is used in PCs with BIOS and is currently replaced by the GPT in UEFI PCs. On USB mass storage devices mostly the MBR is applicable.

The Master Boot Record

The MBR occupies 512 bytes at the beginning (LBA zero) of the medium. The first 446 bytes can store executable code the BIOS executes when a PC is booting. The executable code is then responsible for booting from an bootable partition. Beginning with byte 446 the partition table starts. There is place for up to four partition table entries. The last two bytes are the boot signature which identify the MBR. Byte 510 must be 55h and byte 511 must be AAh[14].

Since there is only place for four partition table entries, a disk formatted with the MBR normally could only have up to four different partitions, called primary partitions. If more partitions are required, extended partitions may be used. An extended partition has its own partition table enclosed by an extended boot record (EBR). The partition table entry in the MBR then points to the EBR which is followed by the extended partition. An EBR can contain one additional entry for another extended partition, thus there is practically no limit to the number of extended partitions[2].

Table 5.1.: Partition table entry in the MBR, compare to: [2]

Byte	Size	Description
0	1	Boot Indicator, 00h for non bootable partition, 80h for bootable partition
1	1	CHS addressing
2	2	CHS addressing
4	1	Partition Type
5	1	CHS addressing
6	2	CHS addressing
8	4	LBA of the first sector
12	4	Total number of sectors/blocks the partition occupies

The important fields of a partition table entry in the MBR are located at byte four, eight and twelve. The fields for CHS addressing are pretty much obsolete. It is a former way to address the blocks on a block device through cylinder, head and sector information. Today only the logical block addressing mechanism is important[18].

The partition table entry holds the first logical block address of the partition. This is where the partition starts, and the content of the file system of the desired partition begins. The entry also holds the total number of blocks the partition occupies, but in most cases this value is also stored in the file system of the partition, and most software uses this value instead[2].

The partition type can either be a value indicating which file system the partition has, e.g. 0Bh or 0C for FAT32, or a hint for an extended partition. Extended partitions are identified by values of 05h or 0Fh. If the partition table entry is unused the partition type must equal to zero[2].

All fields bigger than one byte are stored using little endian style, as in the FAT32 file system.

Drives without a partition table

Sometimes drives do not have a partition table. If the device requires only one partition, the partition table is only a waste of space. If a device does not have a partition table, the file system directly begins at LBA zero.

But it is pretty hard to determine if the device has a partition table or if the file system directly starts, thus most devices have a partition table[2].

5.2. The FAT32 file system

The FAT32 file system was developed and published in 1996 with Windows 95. FAT means File Allocation Table, which is an important part of every FAT32 system, but more on that later. There are two ancestors of the FAT32 file system, FAT12 and FAT16. They vary in the size of an entry in the FAT and other fields. The entries of the FAT in a FAT32 file system have the size of 28 bit. Due to the 32 bit length field of a file, the file size is limited to 4 GiB - 1 Byte, which is sometimes unpleasant in everyday situations[16].

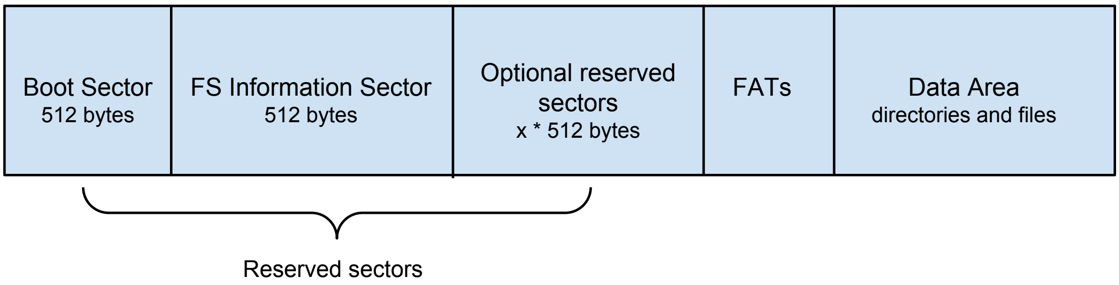
5.2.1. General layout

The general layout of every FAT32 file system consists of following parts[4, 16]:

1. Reserved Region (Boot Sector, FS Information Sector, optional reserved sectors)
2. File Allocation Table
3. Data area (directories and files)

Figure 5.1 illustrates the general layout from the beginning of a FAT32 formatted volume to its end. The sector size is assumed to be 512 bytes, which is mostly the case. Nevertheless, it can be different from 512 bytes. The sector size can be determined from the boot sector structure at the beginning of the medium. Directly after the reserved region the file allocation tables are located. The last and also biggest region is the data area. In the data area the actual content, like directories and files, is saved. Every part is described in detail in the following sections.

Figure 5.1.: General layout of an FAT32 formatted volume



Reserved Region

The reserved region includes the boot sector and the FS Information sector. The boot sector holds important information of the FAT32 file system. For example the sector and the cluster size, the start cluster of the root directory and how many FATs exist. The FS Information Sector holds information about the last allocated cluster and the free cluster count. A file system driver can then easier locate a free cluster and give information about the remaining free space on the disk. After that optional reserved sectors can follow, for example a backup of the boot sector and the FS Information sector[2].

File Allocation Table

The contents of the FAT32 file system is structured in so called clusters. A cluster has a specific size which is specified in the boot sector. A cluster is the smallest unit which can be allocated in the file system. That means if a file for example only has one byte of content but the cluster size is 4096 bytes the file needs 4096 bytes of space. The unneeded 4095 bytes are padded and wasted because they cannot be used for other files.

Every FAT32 file system has a definite amount of clusters which can be used for storing contents. The FAT gives an overview of which clusters are used and which are free. The FAT is a dynamically linked list. Giving a start cluster the FAT can be followed up to the end, identified by an end of chain mark. The resulting cluster chain helps locating the contents on the disk.

Data area

The data area holds the actual content, of the file system, the user wants to save. These are directories and files.

5.2.2. Boot Sector and FS Information Structure

Boot Sector

At the beginning of every FAT32 file system the boot sector is located. Table 5.2 shows the contents of the boot sector. The boot sector has the size of one sector (BPB_BytsPerSec), which is typically 512 byte. Only the fields of interest are shown. A complete overview is given in the official FAT specification from Microsoft[4].

Table 5.2.: Boot Sector, compare to: [2, 4]

Name	Offset	Size	Description
BPB_BytsPerSec	11	2	Count of bytes per a single sector. This is mostly 512 bytes, other allowed values are 1024, 2048 and 4096. A lot of file system drivers assume that this field is 512 and do not check it!
BPB_SecPerClus	13	1	Count of sectors per cluster. This value must be a power of two greater than 0 and is mostly 8. The cluster size in bytes can be calculated with BPB_BytsPerSec * BPB_SecPerClus.
BPB_RsvdSecCnt	14	2	Number of reserved sectors at the beginning of the volume, including the boot sector, preceding the FATs. This value is typically 32.
BPB_NumFATs	16	1	The number of FATs in this file system. The FATs can be mirrored to provide redundancy to ensure that there is always a valid FAT which is not corrupt due to bad sectors or something else. This value is typically 2, meaning there are two different FATs holding the same information.
BPB_TotSec32	32	4	The total amount of sectors in the file system.
BPB_FATSz32	36	4	The number of sectors one FAT occupies.
BPB_ExtFlags	40	2	Bit 0-3: Zero based number of the valid FAT if mirroring of FATs is disabled. Bit 7: Indicates if FATs are mirrored or not. 0 for mirroring, 1 if only one FAT is valid. Valid FAT can be determined with Bit 0-3. Other bits are reserved.
BPB_RootClus	44	4	The start cluster of the root directory. This is typically cluster 2.
BPB_FSInfo	48	2	The sector number of the FS Information Sector in the reserved region.
BPB_BkBootSec	50	2	If this value is non zero, it indicates the sector number of the backup boot sector within the reserved region. This value is typically 6.
BS_VolLab	71	11	Human readable string which gives the volume a name. This field is often replaced by a volume label entry in the root directory.

The boot sector has the same boot signature at byte 510 and 511 like the MBR. That is another problem why it is problematic determining if the current drive has an MBR or if the file system starts directly.

FS Information Structure

The FS Information Structure, also called FSInfo Sector Structure in Microsoft documents, helps finding free clusters quickly. Because the FAT can be very big in a FAT32 file system it can take a lot of time to go through the whole FAT to search for a free cluster. For that reason, the FSInfo Structure holds a hint to the last allocated cluster. It also stores the count of free clusters. The location of the sector is stored in the boot sector.

Table 5.3.: FSInfo Sector, compare to: [4]

Name	Offset	Size	Description
FSI_LeadSig	0	4	Fixed value of 41615252h to identify the FSInfo Sector.
FSI_Reserved1	4	480	Reserved and normally set to zero.
FSI_StrucSig	484	4	Fixed value of 61417272h to identify the FSInfo Sector.
FSI_Free_Count	488	4	The amount of free clusters in the volume. If FFFFFFFFh the free cluster count is unknown and should be computed.
FSI_Nxt_Free	492	4	The last known allocated cluster in the volume. A file system driver should use this to start searching for the free clusters. This does not mean the cluster after the last known is really free, it is just a hint! If FFFFFFFFh the last allocated cluster hint is unknown and the whole FAT has to be searched.
FSI_Reserved2	496	12	Reserved and normally set to zero.
FSI_TrailSig	508	4	Fixed value of AA550000h to identify the FSInfo Sector.

5.2.3. File Allocation Table

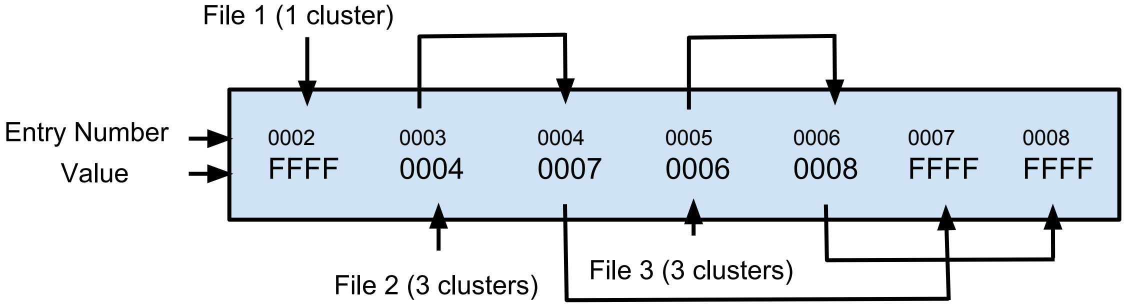
As already mentioned, the File Allocation Table (FAT) is a dynamically linked list with a fixed size. It starts directly after the reserved region. The size of the FAT is stored in the boot sector of the FAT32 file system. The FAT is very important, because it holds the information about which clusters are used and which one not. Moreover it defines which different clusters combined define a file resp. directory. Thus the FAT can be mirrored, for backup reasons. The total count of FATs is located in the boot sector. The next FAT directly begins after the current FAT. Normally on a FAT32 file system there are two FATs holding the same information.

Every entry of the FAT is 32 bit, with the lower 28 bit representing the cluster number. The other four bits are unused. The first two entries in the FAT do not store any information about clusters, the data area begins with cluster 2[2].

Cluster chains

A series of clusters is called a cluster chain. Every directory or file consists of a specific cluster chain defining the location of the content on the disk. To follow such a chain, a start cluster is needed. The start cluster of the root directory for example is stored in the boot sector. A FAT32 file system then seeks into the FAT where the start cluster is located. In bytes this is 4 * start cluster (Every entry is 32 bit = 4 byte) from the beginning of the FAT. The value stored at this location is the next cluster in the chain. This is repeated until the value stored is an "end of chain mark". This determines the end of a cluster chain. A value above FFFFFF7h is an end of chain mark. A value of zero indicates that the cluster is free and a value of one that the cluster is reserved. A value of FFFFFF7h stands for a bad cluster.

Figure 5.2.: Simplified illustration of a FAT and following cluster chains



After evaluating the cluster chain, the file system driver can access the contents of the clusters in the data area. To do this, the logical block address where a cluster starts can be computed as follows: ((cluster - 2) * sectors per cluster) + data area offset

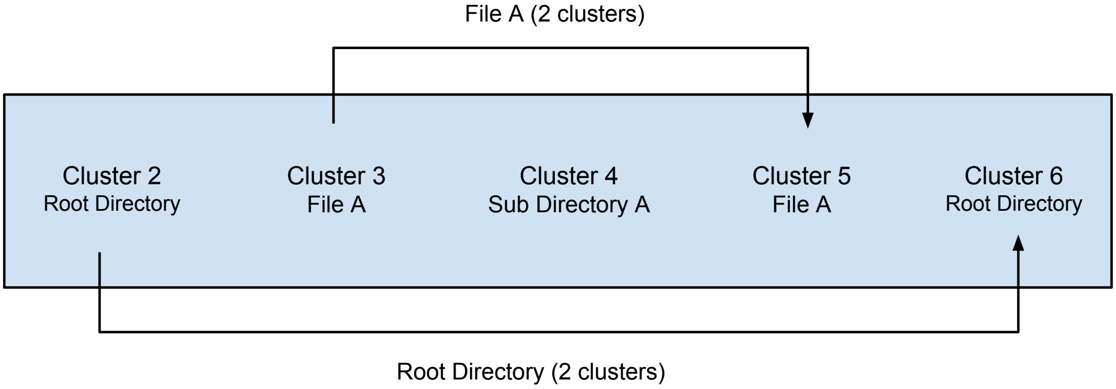
5.2.4. Directories and files

Directories and files are located in the data area of a FAT32 file system. The exact location of the contents and which clusters correspond to the directory or file must be determined by a cluster chain from the FAT. A directory consists of multiple 32 byte entries describing the contents of the directory. The contents can either be other directories, so called sub-directories or files. The root directory is always present on a FAT32 file system. The start cluster of the root directory is stored in the boot sector.

Files do not have a defined structure, unlike directories. The raw data the file consists of is located in the different clusters, just as it is. The only thing defined, is the order, which can be determined from the FAT.

Figure 5.3 shows a simplified example of the data area. The root directory starts at cluster 2 and ends with cluster 6. There is also a file in the data area, consisting of cluster 3 and 5. In cluster 4 a sub directory is located consisting of one cluster only. The whole data area is separated into clusters with a certain size, containing either chunks of data of a directory or a file.

Figure 5.3.: Simplified illustration of the content in the data area



Fat Directory Entry

The following table describes the structure of a Fat Directory entry. Every entry, normally, represents either another (sub)directory or file. The root directory can also have a special entry not only describing files and other directories but an optional volume label which gives the volume a name. Such a volume label could also be specified in the boot sector, but it is more common to specify it in the root directory. Another special entry is the long file name entry, which is described later.

Table 5.4.: Fat Directory Entry, compare to: [2, 4]

Name	Offset	Size	Description
DIR_Name	0	11	The short name of the directory or file.
DIR_Attr	11	1	Attributes to determine if this entry describes a directory or a file, etc. Described in table 5.5
DIR_NTRes	12	1	Reserved.
DIR_CrtTimeTenth	13	1	Timestamp which holds the tenth of a second when the file was created. The range is from 0 to 199.
DIR_CrtTime	14	2	The time the file was created.
DIR_CrtDate	16	2	The date the file was created.
DIR_LstAccDate	18	2	The date the file was last accessed (read or write).
DIR_FstClusHI	20	2	The higher two bytes of the entry's start cluster. Unused if the entry is not a file or directory.
DIR_WrtTime	22	2	The time the file last modified (write).
DIR_WrtDate	24	2	The date the file last modified (write).
DIR_FstClusLO	26	2	The lower two bytes of the entry's start cluster. Unused if the entry is not a file or directory.
DIR_FileSize	26	4	The length of a file in bytes. Unused if the entry is not a file.

Short name

Every entry has a short name which is a human readable name. The name consists of eight characters for the file name and three optional characters for the file extension. Therefore it is also called 8.3 name. The period to separate name and extension is not stored in the short name. If a name does not need the whole eight characters the unused characters have to be padded with spaces (ASCII: 20h). The same applies for the extension. A short name has various limitations[2]:

- Each character is only eight bit (no unicode support)
- It has to begin with a letter or a number
- Every character is stored upper case
- The only allowed special characters are: $ % ′ - _ @ ∼ ‘ ! ( ) ˆ # &

If the first byte of the short name equals to E5h, then this directory entry is free and has been deleted. If the first byte equals to 00h, then this entry is also free, but there are not any following entries, looping through the entries can thus be stopped. In the first case it can happen that there are valid entries that follow. If the first byte equals to 05h, then the actual value of the first byte shall be E5h, which is a KANJI lead byte used in Japan. This is a workaround for avoiding the entry to be accidentally treated as free (deleted)[4].

Attributes

Table 5.5.: Attributes of an entry, compare to: [2, 4]

Name	Value	Description
ATTR_READ_ONLY	01h	The file is read only and writing to it should fail.
ATTR_HIDDEN	02h	A hidden entry the user should only see when explicitly asking for it.
ATTR_SYSTEM	04h	A file from the operating system.
ATTR_VOLUME_ID	08h	This entry is not a file and not a directory, it is the entry which stores the volume label. This can occur only in the root directory and only once.
ATTR_DIRECTORY	10h	Indicates that the entry describes a (sub)directory.
ATTR_ARCHIVE	20h	Value helping backup utilities identifying files that changed since the last backup. Should be set if a file has been changed.
ATTR_LONG_NAME	0fh	Indicates that this entry is not a file or a directory but an entry which holds a long file name.

The last attribute in table 5.5 stands for a long file name entry. Because of the limitations of the short name, Microsoft added support for long file names afterwards. To ensure backward compatibility they used some sort of work around to hide the long file names in normal directory entries. They are discussed later in detail.

If the volume id attribute is set, the short name is not actually the name of a file, it is the name of the volume. The volume name is shown in the Windows Explorer directly left of the drive letter. The volume label can have up to eleven characters, but there is no period between the eighth and the ninth character[2, 4].

Date and time fields

In the directory entry there are many date and time fields, the following describes how the date and time is stored in these fields. Every date and time field consists of two bytes.

Table 5.6.: Date format, compare to: [2, 4]

Bits	Description
0-4	Day of month, range 1-31
5-8	Month, starting with 1 for January
9-15	Count of years from 1980, range 0-127 corresponding to 1980-2107

Table 5.7.: Time format, compare to: [2, 4]

Bits	Description
0-4	Count of seconds with a resolution of 2 seconds, range 0-29 meaning 0-58 seconds
5-10	Minutes, range 0-59
11-15	Hours, range 0-23

5.2.5. Subdirectories

Every subdirectory has two special entries. The dot (.) and the dotdot (..) entry. The dot entry points the the current subdirectory itself. It has the same values as the entry for the subdirectory in the parent directory, i.e. same date and time fields, same start cluster, etc. The dotdot entry points to the parent directory, but the date and time fields remain like in the subdirectory. The start cluster of the dotdot entry is the same as for the parent directory except if the parent directory is the root directory, then it is set to zero[4].

These two entries are the only exceptions where a short name starts with periods. The dot and dotdot entry must not have preceding long file name entries[4]!

The root directory does not have these two special entries, this rule applies for subdirectories only[4]!

5.2.6. Long File Name entries

As already mentioned the short name of a directory entry has several disadvantages. For that reason Microsoft afterwards faced this problem by adding long file name (LFN) entries. These entries ”hide” in the normal directory entries and look like a hidden file with a start cluster of zero, indicating that the file does not occupy any space. Thus old implementations which do not support long file name entries can nevertheless work with the file system and the 8.3 names.

Long file names can have up to 255 characters and allow upper and lower case characters, leading, trailing and multiple periods and spaces in a file name[2]. Additionally these special characters are allowed[4]: + , ; = [ ]

Each entry with a normal short name, can have one or more long file name entries preceding the actual entry. A long file name entry can store up to 13 unicode characters, i.e. every character consists of two bytes instead of eight like in the short name.

Table 5.8.: Fat LFN Directory Entry, compare to: [2, 4]

Name	**Offset**	**Size**	**Description**
LDIR_Ord	0	1	The number (order) of the entry in the sequence of long file name entries.
LDIR_Name1	1	10	The first five characters of the long file name.
LDIR_Attr	11	1	Must be ATTR_LONG_NAME.
LDIR_Type	12	1	Must be zero.
LDIR_Chksum	13	1	The checksum of the short name associated with this long file name.
LDIR_Name2	14	12	The next six characters of the long file name.
LDIR_FstClusLO	26	2	Must be zero. Long file name entries do not have a start cluster.
LDIR_Name3I	28	4	The last two characters of the long file name.

The first byte is the number of the LFN entry. The first LFN entry before the short name entry, must have a value of one in that field. The second a two, and so on. If the LFN entry is the last entry, bit six of the LDIR_Ord field has to be set to one, to indicate that it is the last entry. There can be at most 20 LFN entries preceding a normal entry. The first LFN entry directly before the short name entry represents the begin of the long file name. That means the long file name is stored in ”reverse order” on the disk.

The long file name entry should be terminated with a null character (0h) if there is enough space, and unused characters shall be padded with FFFFh[4].

Checksum

Every long file name entry needs a checksum of the corresponding short name. This checksum is calculated as follows:

Listing 5.1: Calculation of short name checksum in C [4]

```
//-----------------------------------------------------------------
// ChkSum()
// Returns an unsigned byte checksum computed on an unsigned byte
// array. The array must be 11 bytes long and is assumed to contain
// a name stored in the format of a MS-DOS directory entry.
// Passed: pFcbName Pointer to an unsigned byte array assumed to be 11
   bytes long.
```

```
// Returns: Sum An 8-bit unsigned checksum of the array pointed to by
    pFcbName.
//----------------------------------------------------------------------
unsigned char ChkSum (unsigned char *pFcbName)
{
   short FcbNameLen;
   unsigned char Sum;

   Sum = 0;
   for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--) {
      // NOTE: The operation is an unsigned char rotate right
      Sum = ((Sum & 1) ? 0x80 : 0) + (Sum >> 1) + *pFcbName++;
   }
   return (Sum);
}
```

Generating short names

Every directory entry which has preceding long file name entries nevertheless needs a valid short name. There exist different algorithms for generating valid short names given a long file name. These algorithms mainly consist of removing illegal characters and convert them to underscores (_), removing spaces and periods, truncating the long file name to eight and the extension, if available, to three characters. All characters must be converted to upper case and a tilde (~) must be added if the file name was truncated, contained illegal characters or an entry with the same short name already exists in the directory. The official FAT specification[4] and the book from Jan Axelson[2] describe two different approaches to generate valid 8.3 short names.

Part II.

Implementation

6. Purpose and Overview

Purpose

The developed Android framework for accessing USB mass storage devices shall meet determinate requirements:

- The mass storage device is accessible without root rights
- The API provides methods for enumerating through all connected mass storage devices and their partitions
- The API is easy to use and orientates towards the java.io.File API
- The framework provides basic features like: Adding and removing directories and files, read and write access to files, moving directories and files located on the same volume
- The framework shall be licensed under the Apache License, Version 2[1]

The framework shall at least support *bulk-only transport* mass storage devices, which are using the SCSI transparent command set. It shall support devices which are formatted with the MBR partition table and the FAT32 file system. Despite the framework a simple example application shall be developed also to test and demonstrate the framework. The framework and the example application are publicly available at github[2].

The following sections describe the relevant parts of the framework, but not every single detail. To get an insight how the whole thing works, the source code, which is well documented, may be examined directly.

Overview

This sections gives a brief overview over the whole framework, in the following chapters important parts of the framework are discussed in detail.

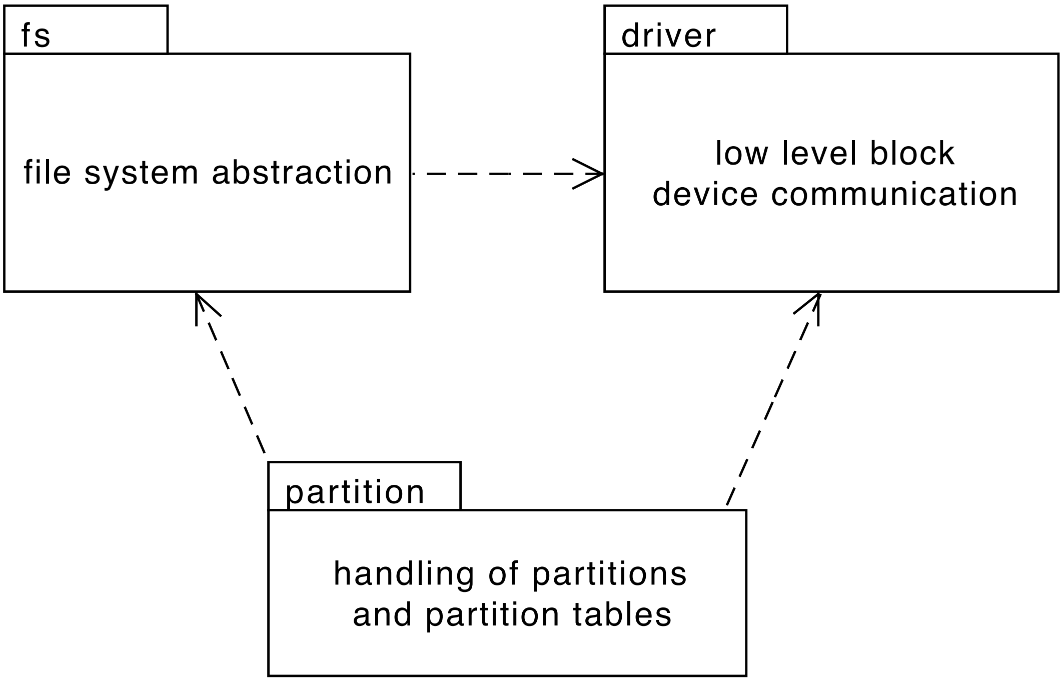
The framework and the example application are written in Java. This is Android's main language for developing applications. The framework uses the standard Java API and the Android USB host API to access USB devices. The example application uses the API of the developed framework and the standard Android API for creating user interfaces.

The framework can roughly be structured in three parts, figure 6.1 shows a UML package diagram of the framework. The packages in the UML diagram correspond to the Java packages in the source code. Please note that the UML diagrams in this thesis are often simplified and do not cover all details.

[1] `http://www.apache.org/licenses/LICENSE-2.0.html`

[2] `https://github.com/mjdev/libaums`

Figure 6.1.: Package overview of the framework



Driver

The driver package is responsible for the low level communication with the block device over USB. It uses USB bulk transfers to access and to communicate with the USB device. It contains the SCSI commands described in the theory part (section 4.3). The package provides methods for reading and writing raw data from and to the device storage.

Partition

This package is relevant for handling partition tables and recognizing the different partitions and their file systems on a mass storage device. Thus it needs direct access to the block device, as well as access to the file system implementations. The partition package contains code for handling MBR partition tables.

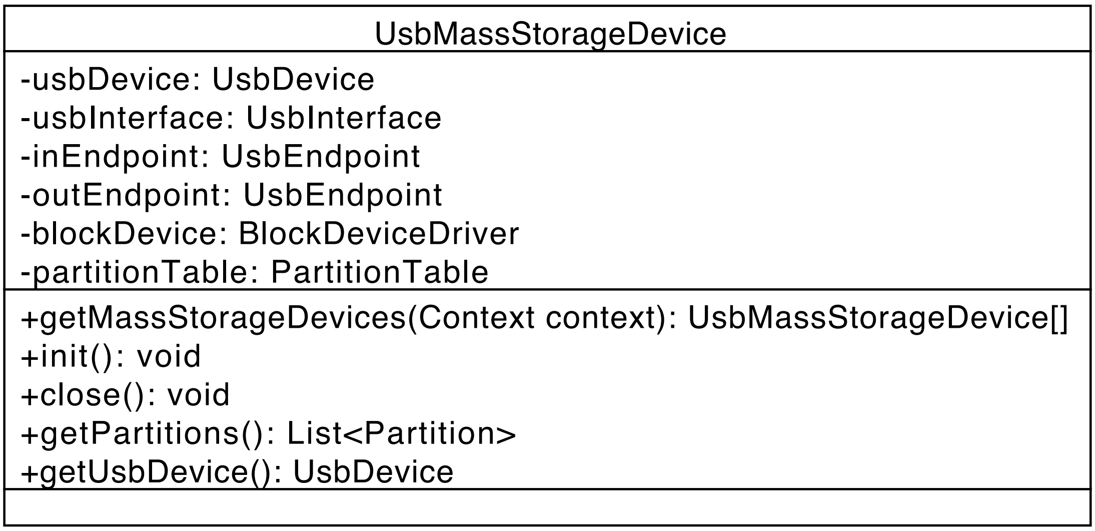
File system

The fs package contains the code for the FAT32 file system. It needs direct access to the block device's raw data, in particular to the raw data of the specific partition it represents. That means it only has indirect access to the driver package. All method calls to read or write raw data are routed through the partition package, to handle the different partitions on a block device correctly.

6.1. Using the Framework

The UsbMassStorageDevice class is the main entry point for accessing mass storage devices. It provides a static method which returns all available mass storage devices. This method loops through all connected USB devices and checks if the connected device is a valid device following the USB mass storage class. The mass storage devices can then be initialized, via the init() method. The initialization process consists of reading the partition table, creating the corresponding partitions and evaluating the desired file system for each partition. The available partitions can then be accessed easily via a getter. The close method closes the USB communication and releases the USB Interface.

Figure 6.2.: UML class diagram of the UsbMassStorageDevice


UsbMassStorageDevice
-usbDevice: UsbDevice
-usbInterface: UsbInterface
-inEndpoint: UsbEndpoint
-outEndpoint: UsbEndpoint
-blockDevice: BlockDeviceDriver
-partitionTable: PartitionTable
+getMassStorageDevices(Context context): UsbMassStorageDevice[]
+init(): void
+close(): void
+getPartitions(): List<Partition>
+getUsbDevice(): UsbDevice

The class also has a lot of private members for communicating with the USB device via the Android API. There is a getter for the underlying UsbDevice, mainly for requesting the permission for communication by the user. Requesting permission is described in chapter 3, which is about the Android USB host API.

Partition

The class Partition represents a single volume on a mass storage device. It provides a getter for the volume label and a getter for the file system to access the contents of the partition.

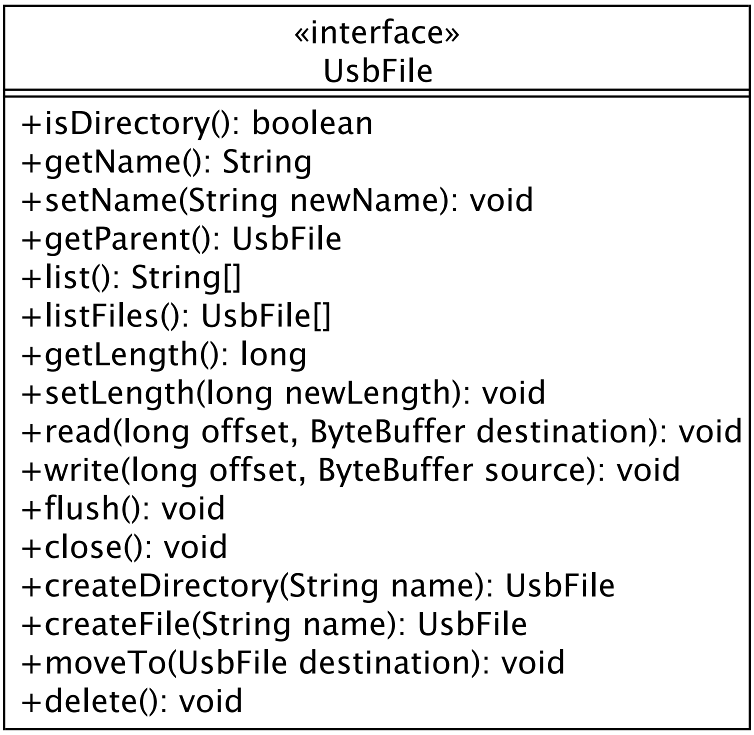
FileSystem

The FileSystem interface provides a getter for the volume label, which returns exactly the same string like the getter in the class Partition. In fact the getter of the Partition class simply delegates the call to the FileSystem class. Another important method allows accessing the root directory of the file system.

UsbFile

The UsbFile interface represents an abstraction for files and directories. Every directory or file is an UsbFile. The root directory returned by the FileSystem interface is also a UsbFile. The UsbFile interface provides various methods to access and modify the contents of a file or directory. A complete documentation on every method offers the javadoc in the source code. Note that some methods only make sense for directory or files, but not for both of them!

Figure 6.3.: UML class diagram of the UsbFile interface


«interface»
UsbFile
+isDirectory(): boolean
+getName(): String
+setName(String newName): void
+getParent(): UsbFile
+list(): String[]
+listFiles(): UsbFile[]
+getLength(): long
+setLength(long newLength): void
+read(long offset, ByteBuffer destination): void
+write(long offset, ByteBuffer source): void
+flush(): void
+close(): void
+createDirectory(String name): UsbFile
+createFile(String name): UsbFile
+moveTo(UsbFile destination): void
+delete(): void

Code example

The following code demonstrates the use of the classes introduced above. The example simply takes the first USB mass storage device which was found and lists the contents of the root directory of the first partition.

Listing 6.1: Code example for accessing the contents of a mass storage device

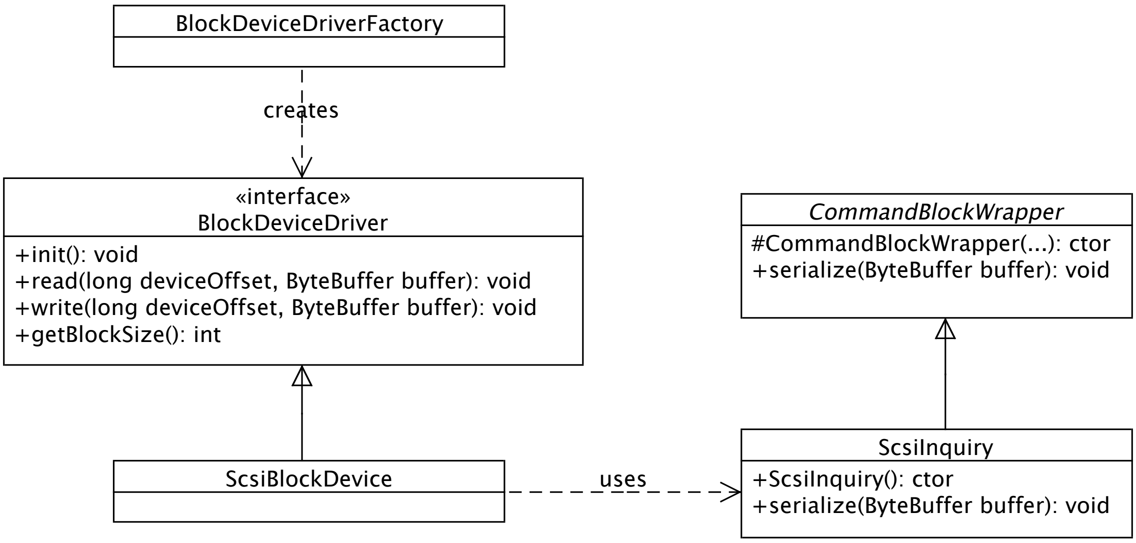
```
private void setupDevice() {
   // the getter needs a Context (Activity or Service) as parameter
   UsbMassStorageDevice[] devices =
      UsbMassStorageDevice.getMassStorageDevices(this);
   if(devices.length == 0) {
      Log.w(TAG, "no device found!");
      return;
   }

   UsbMassStorageDevice device = devices[0];

   try {
      // before initializing the device, user must grant permission to
          communicate
      // this can be done with the UsbManager class and a
          BroadcastReceiver like shown in the
      // section about the Android USB host API
      device.init();

      // always use the first partition of the device
      FileSystem fs = device.getPartitions().get(0).getFileSystem();
      Log.d(TAG, "volume label: " + fs.getVolumeLabel());

      UsbFile root = fs.getRootDirectory();
      String[] contents = root.list();
      for(String str : contents) {
         Log.d(TAG, str);
      }
   } catch (IOException e) {
      Log.e(TAG, "error setting up device", e);
   }
}
```

7. Inside the packages

7.1. The driver package

As mentioned previously, the driver package is responsible for handling the low level communication with the block device. It can access the USB device via bulk IN and OUT transfers. Currently only a block device driver for the SCSI transparent command set is available.

Figure 7.1.: UML class diagram of the driver package


BlockDeviceDriverFactory
creates
«interface»
BlockDeviceDriver
+init(): void
+read(long deviceOffset, ByteBuffer buffer): void
+write(long deviceOffset, ByteBuffer buffer): void
+getBlockSize(): int
CommandBlockWrapper
#CommandBlockWrapper(...): ctor
+serialize(ByteBuffer buffer): void
ScsiBlockDevice
uses
ScsiInquiry
+ScsiInquiry(): ctor
+serialize(ByteBuffer buffer): void

BlockDeviceDriver

The BlockDeviceDriver interface is a general representation of a block device. It provides methods for reading and writing raw data from and to the device's media storage. It takes a device offset and a ByteBuffer as parameter to determine the offset of a read or write. The ByteBuffer indicates the length of the data which shall be read or written. If data shall be read, the data is read into the ByteBuffer, otherwise the data in the ByteBuffer is written to the device. It also offers a getter to determine the block size of the connected device.

BlockDeviceDriverFactory

This class is in charge of creating a suitable block device driver for the connected mass storage device. Currently it always creates a ScsiBlockDevice. No other driver is currently supported. This class is intended to make further development and integration of other device drivers easier. There are also factory classes in the two other packages for creating suitable partition tables and file systems.

ScsiBlockDevice

This is the representation of a block device driver which uses the SCSI transparent command set for communicating with devices. It transfers SCSI commands to the device, receives the desired responses from the device and interprets them.

All SCSI commands a modeled in an own class which extend the CommandBlockWrapper. The CommandBlockWrapper is an abstract class which is always coupled with a SCSI command. As already mentioned, every SCSI command is enclosed by a CBW in the SCSI transparent command set protocol. The CBW offers a method to serialize itself to a ByteBuffer. This data can then be transmitted directly to the device. The serialized data include the direction of the command, the transfer length in the transport phase and the length of the SCSI command.

Every SCSI command also offers the serialization to a ByteBuffer, it first calls the serialization method of the CBW class (super class) and then adds the own data to the ByteBuffer. Using this approach it is easy to wrap the CBW around the SCSI commands and new commands are straightforward to implement.

In the UML diagram 7.1 only the SCSI INQUIRY command is shown, but there are, of course, also classes for all other commands presented earlier.

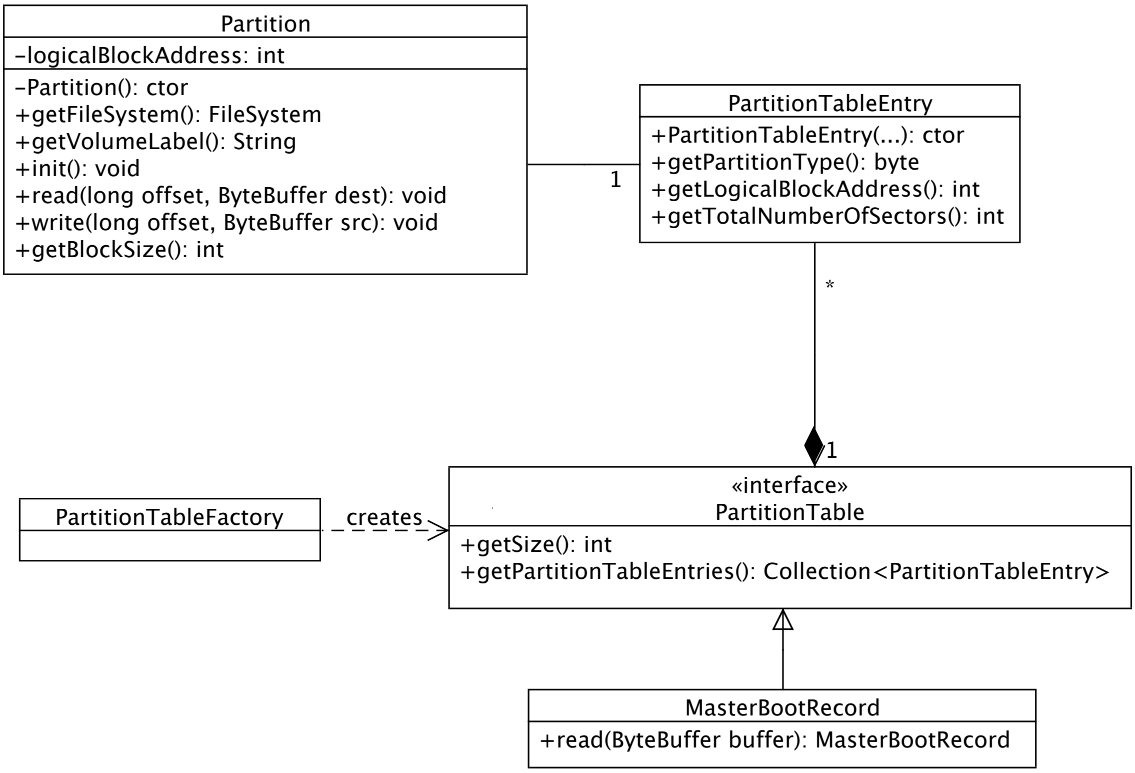
7.2. The partition package

The partition package is responsible for handling the partition table on a USB mass storage device. Currently only the MBR partition table is supported. Determining the partition table can be pretty hard, because there is no hint which type of partition table is stored on the device. Therefore the data at LBA zero has to be read (normally a partition table starts at the beginning of a volume) and it has to be checked if the data represents a valid partition table. Figure 7.2 illustrates the contents of the partition package. There is again a factory class for creating suitable partition tables.

PartitionTable

This interface represents in general a partition table. It provides a getter to receive all partition table entries in the table. There is also a method for getting the size of the partition table. For the MBR this is 512 bytes. The factory class uses this size, to determine how many bytes from the mass storage device have to be read.

Figure 7.2.: UML class diagram of the partition package


Partition
-logicalBlockAddress: int
-Partition(): ctor
+getFileSystem(): FileSystem
+getVolumeLabel(): String
+init(): void
+read(long offset, ByteBuffer dest): void
+write(long offset, ByteBuffer src): void
+getBlockSize(): int
1
PartitionTableEntry
+PartitionTableEntry(...): ctor
+getPartitionType(): byte
+getLogicalBlockAddress(): int
+getTotalNumberOfSectors(): int
*
1
PartitionTableFactory
creates
«interface»
PartitionTable
+getSize(): int
+getPartitionTableEntries(): Collection<PartitionTableEntry>
MasterBootRecord
+read(ByteBuffer buffer): MasterBootRecord

PartitionTableEntry

The PartitionTableEntry represents the information of a partition stored in the partition table. It saves the logical block address where the partition starts, the total number of sectors/blocks the partition occupies and the type of the partition. This is mostly the file system type of the partition, but in case of the MBR this can also be an extended partition.

MasterBootRecord

This class covers the Master Boot Record implementation. It has a static read method which returns an instance of the MasterBootRecord class, or null if the data in the ByteBuffer does not look like a Master Boot Record.

Partition

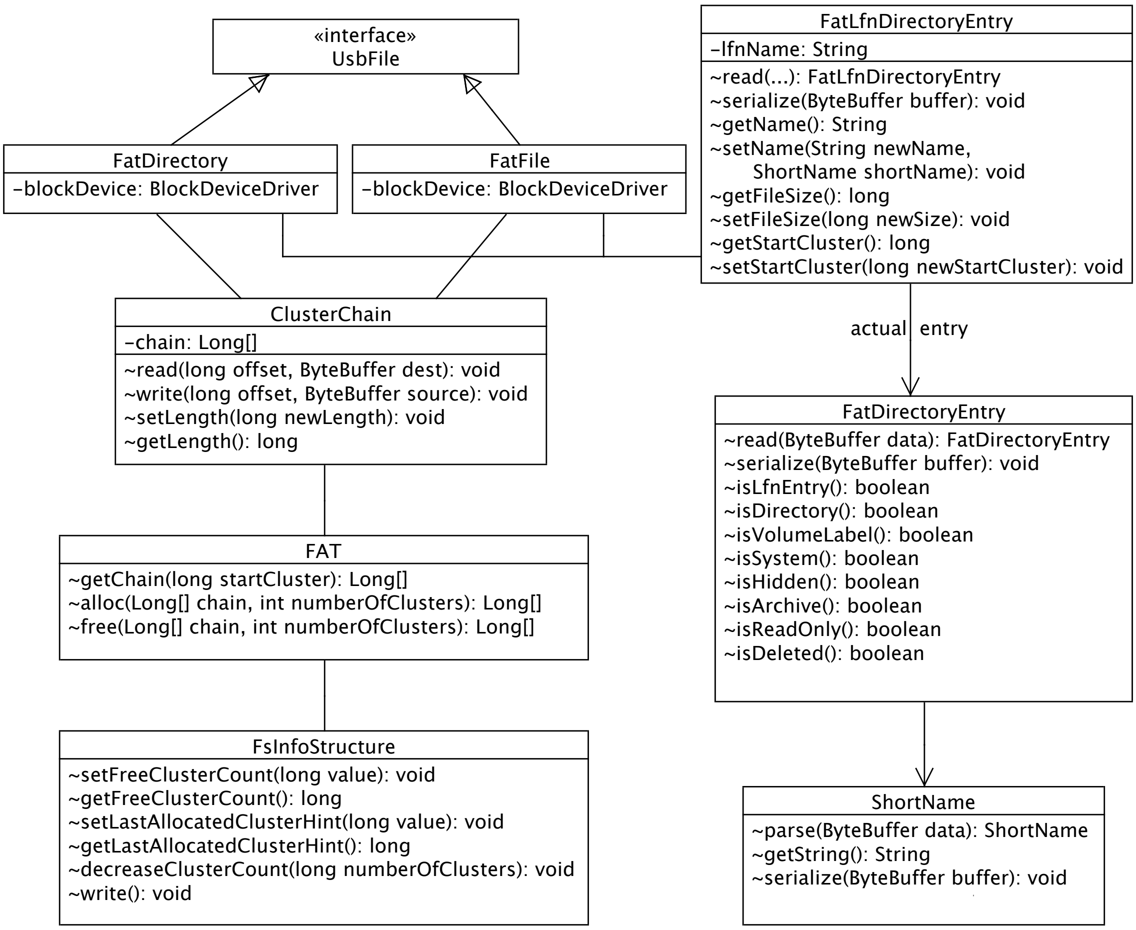
The Partition class was already introduced in the overview section (6), but this time the focus lies on the interaction with other classes of the package and the file system package.

The Partition class has access to the PartitionTableEntry it represents. It uses the information stored in the entry to determine the starting point (LBA) of the partition and to initialize a suitable file system, the user can then access. The Partition class implements also the BlockDeviceDriver interface from the driver package, described earlier. This is needed because every partition represents an independent volume of the mass storage device. A file system driver should be independent of the location of a partition, hence the partition is responsible for translating the requests of the file system driver according to the starting point, the logical block address, of the partition.

7.3. The file system package

The main classes to access the file system and the contents of it, are the previously mentioned FileSystem class, and the UsbFile interface. These two classes are not explained again, instead the focus lies on a deeper insight into the implementation of the FAT32 file system.

Figure 7.3.: UML class diagram of the FAT32 implementation


«interface»
UsbFile
FatDirectory
–blockDevice: BlockDeviceDriver
FatFile
–blockDevice: BlockDeviceDriver
FatLfnDirectoryEntry
–lfnName: String
~read(...): FatLfnDirectoryEntry
~serialize(ByteBuffer buffer): void
~getName(): String
~setName(String newName, ShortName shortName): void
~getFileSize(): long
~setFileSize(long newSize): void
~getStartCluster(): long
~setStartCluster(long newStartCluster): void
ClusterChain
–chain: Long[]
~read(long offset, ByteBuffer dest): void
~write(long offset, ByteBuffer source): void
~setLength(long newLength): void
~getLength(): long
actual entry
FatDirectoryEntry
~read(ByteBuffer data): FatDirectoryEntry
~serialize(ByteBuffer buffer): void
~isLfnEntry(): boolean
~isDirectory(): boolean
~isVolumeLabel(): boolean
~isSystem(): boolean
~isHidden(): boolean
~isArchive(): boolean
~isReadOnly(): boolean
~isDeleted(): boolean
FAT
~getChain(long startCluster): Long[]
~alloc(Long[] chain, int numberOfClusters): Long[]
~free(Long[] chain, int numberOfClusters): Long[]
FsInfoStructure
~setFreeClusterCount(long value): void
~getFreeClusterCount(): long
~setLastAllocatedClusterHint(long value): void
~getLastAllocatedClusterHint(): long
~decreaseClusterCount(long numberOfClusters): void
~write(): void
ShortName
~parse(ByteBuffer data): ShortName
~getString(): String
~serialize(ByteBuffer buffer): void

FatDirectory and FatFile

The FatDirectory and FatFile classes are responsible for the directory and file handling. They have two important attributes, the ClusterChain and the FatLfnDirectoryEntry. The FatLfnDirectoryEntry contains the long file name of the directory or file, the start cluster and if it is a file, the length of a file. It also contains information about the date and time the directory or file was created, last accessed and modified. The ClusterChain is responsible for accessing the data located on the disk. A FatDirectory parses and writes the directory entries located in the chain, while the FatFile passes the functionality of reading or writing directly to the user via the UsbFile interface.

FatLfnDirectory and FatDirectoryEntry

The FatLfnDirectory provides methods to access a long file name entry. It encapsulates the long file name as well as the actual entry holding important information about the entry, such like the start cluster or file size, and the short name. The FatLfnDirectory entry mostly delegates the calls which do not have to do with the long file name, such as the getting or setting the start cluster or file size, to the actual entry which is an attribute of the FatLfnDirectory class. Both classes and the ShortName class provide convenience methods for parsing/reading and serializing the data.

ClusterChain and FAT

The ClusterChain handles reading and writing from and to cluster chains given a certain start cluster. It offers read and write methods to read and write the raw data from and to the desired clusters on the disk. It also has methods to set and get the length of the cluster chain. The setLength() method, grows and shrinks the cluster chain as needed with the help of the FAT class. When writing to the chain, the cluster chain will also be dynamically increased if needed.

The FAT is responsible for the cluster distribution in the File Allocation Table. The method getChain() returns an array containing all clusters in the chain, including the start cluster at the first position, given a certain start cluster. The FAT class can also allocate new clusters or free unneeded clusters from a chain. When allocating or freeing clusters the FAT class also sets the new information regarding the free clusters and the last allocated cluster hint in the FSInfoStructure.

Fat32FileSystem and Fat32BootSector

These two classes are not shown in the UML class diagram 7.3, because they do not play an important role in the interaction of the classes. But nevertheless, without them the whole system would not work. The Fat32FileSystem class is mainly responsible for initializing the file system, meaning reading the boot sector, preparing the FAT and FSInfoStructure. It is also responsible for initializing the root directory and to hand it (via a getter) to the user, if desired.

The Fat32BootSector class reads the information of the boot sector of the FAT32 file system, and provides getter for important information, other classes need to access. This information includes but is not limited to the cluster and sector size and the start cluster of the root directory.

Part III.

Quality Management

8. Testing

8.1. Overview

To ensure the quality of the Framework it has been tested on a wide range of different Android devices. The framework has also been tested with different USB pen drives and an external HDD, with external power source. Card readers have not been tested! The results of the tests are explained in the following sections.

8.1.1. Testing criteria

On every device following aspects were tested, if they succeeded or not:

- Listing the contents of directories
- Reading and writing from and to files
- Adding directories and files
- Removing directories and files
- Renaming directories and files
- Moving directories and files to other directories
- Writing files bigger than the cluster size, to check if the dynamic growing of a cluster chain works correctly

8.2. Results

Table 8.1 shows the test result for the devices which have been tested. If all aspects work properly, the test is successful.

8.2.1. Native support

Some devices support mounting USB mass storage devices without root rights natively. This was verified for the Samsung Galaxy S3 and the Archos 101 G9. When connecting a mass storage device via the USB OTG adapter to such a device, the mass storage device is automatically mounted and can be either accessed with the file manager which came with the device or any third party file manager. On the Archos device the mass storage is mounted under */mnt/ext_storage*.

Table 8.1.: Test results

Device	Android Version	Success	Comments
Archos 101 G9	4.0.4	Yes	Has native support for USB mass storage devices.
Google Nexus 4	4.4.2	No	Does not have the USB host feature[1].
Google Nexus 5	4.4	Yes	-
Google Nexus 7	4.2.2	Yes	-
Google Nexus 7	4.4.2	Yes	-
Google Galaxy Nexus	4.3	Yes	-
Google Nexus S	4.1.2	No	Does not have the USB host feature.
Samsung Galaxy S3	4.3	Yes	Has native support for USB mass storage devices.

8.2.2. Performance test

On Android versions lower 4.3 a specific method in the API is not available. It was later added with API level 18. To support older Android versions and to overcome this lack, the framework uses a different API call on lower Android versions. This workaround can influence on the performance. To examine the performance, the same file is copied from the mass storage to the internal storage on different Android versions, but on an identical device. More information about the API difference, can be found in appendix C.

The file copied was a video file with a size of 155,883,762 bytes, which is approximately 148.6 megabytes. The two devices were Google Nexus 7 tablets with Android versions 4.4.2 and 4.2.2. The same USB pen drive was used for the tests. The file was copied five times on every device, table 8.2 shows the average copy time.

Table 8.2.: Performance test: Average time of copying same file five times on each device

Android Version	Time in Milliseconds	Time in seconds	Time in minutes
4.2.2	90203.1	~ 90	~ 1.5
4.4.2	100040.6	~ 100	~ 1.6

The performance results are pretty interesting because the device with the lower Android version is definitely faster than the one with the newer one. The difference is about ten seconds! The devices behave contrary as assumed. The reason for this is hard to find, maybe the USB host stack has changed greatly between these two versions. But this would imply that the newer Android version provides an inferior performance consulting the USB host support. Another reason could be that installed applications and running services in the background have a huge impact on the performance. This is also an evidence for the difficulty of creating reasonable performance tests on two different devices, although if they both are of the same model!

Additional performance test

Because of the surprising results another test on the device with Android 4.2.2 was run. The test is exactly as the first test, except that this time, in the first test run the offset to write in the buffer is always zero, the second time it is forced to be none zero. That means that in the second run the workaround is enforced and the buffer has to be copied. Again, more information regarding the API difference can be found in appendix C.

Table 8.3.: Performance test: Average time of copying same file five times on same device

Test Run	**Time in Milliseconds**	**Time in seconds**	**Time in minutes**
Offset zero	84751	~ 85	~ 1.4
Offset non zero	90203.1	~ 90	~ 1.5

This time the result is as expected, meaning the overhead of copying the whole data into a temporary buffer takes (about five seconds) longer.

Part IV.

Results

9. Summary

The goal of the thesis was to develop a framework, for the Android operating system, which allows access to USB mass storage devices. These devices include USB pen drives, card readers or even external HDDs. The framework allows discovering and exchanging data with these devices in terms of directories and files.

The thesis gives an overview of basic aspects relating USB in general and how to use the Android USB host API. A very important aspect is the description of the USB mass storage class, including the bulk-only transfer and the SCSI transparent command set. The most important SCSI commands were also introduced. After that a detailed view on the theory about file systems, in detail the FAT32 file system, follows.

After the theoretical part, the developed framework is described, it's purpose, general structure and most important aspects in detail.

The part about the quality management gives a short overview of the interplay of the framework with different devices and Android versions.

9.1. Current status

The Framework works on every device with Android 3.1 or later, which has hardware and software enabled USB host support. Every Android device which meets these requirements should be supported. The framework currently supports mass storage devices using the bulk-only transport with the SCSI transparent command set. The mass storage device must be formatted with an MBR, located at the beginning of the storage. There is no support for other partition tables. Devices completely without an partition table are also unsupported. The supported file system is FAT32, which is the most commonly used one for USB mass storage devices. Other file systems, even the one from the FAT family, like FAT12 or FAT16, are not supported.

Despite these limitations the framework has all features which had been determined at the beginning of this thesis[1]. In the source code, there are currently some TODOs to mark starting points for minor enhancements. But these points do not compromise the everyday use. Currently there are a lot of debug log messages which help to understand the operation of the framework and may decrease the performance, especially when reading or writing huge amounts of data from or to files.

For example, the SCSI REQUEST SENSE command could be added. Further information about an unsuccessful command can be acquired by it. This is not a serious problem, because it is very rare that a device cannot execute a command successfully. During the development, this constellation occurred only when the commands transfered to the device were incorrect and thus the framework had a bug. For the development of extensions to the framework, the addition of this command may be helpful.

[1] The desired requirements and features are described in section 6.

Nevertheless, the framework is very easy to extend. Most parts operate independent from each other and can be easily be exchanged. Other block device drivers, partition tables or file systems can easily be added without changing unrelated parts. This is why the framework often relies on interfaces instead of particular implementations and often uses factory classes for the initialization of instances.

9.2. Conclusion

Developing Android apps, is surely much fun. The Android API is well structured, organized and easy to understand. This also stays on when developing advanced applications like this one.

Very interesting was the aspect of developing low level things like a block device driver and a file system driver in the Java programming language. Normally these things are done in C, and not in a higher level language like Java. But solely relying on Java and strictly avoiding C code was not a problem at any time, Java did the job very well! This shows that Java is also perfectly capable of bringing the object oriented approach to things located at lower levels in an operating system or the kernel.

The documentation on the USB mass storage class and the SCSI transparent command set is very rare and often complex. It needs a lot of foreknowledge on some topics. Nevertheless after consulting various resources and spending a lot of time reading, the comprehension continuously increases.

10. Outlook

The developed framework is usable perfectly at its current status. As always there is room for further development and features. Integrating other block device drivers and file systems are features that come instantly in mind. Maybe someday someone likes to connect his external CD/DVD drive to his Android device?

But not only other block device and file system drivers are possible. Currently the framework has no intelligent caching mechanisms. However it always writes the changes directly to the storage which may be inefficient sometimes. Reading data, at the moment, is implemented straightforward. The appropriate data is just read from disk, there are no special strategies like reading ahead or guessing what data the user wants to access next. These are all things which are available in every up to date operating system and many people invested a lot of time in efficient strategies for caching, etc. Maybe implementing such techniques increase the user experience.

Another useful extension to the framework could be the integration into the new FileSystem API of Java 7[1]. This would give the user the ability to use the default Java API for accessing files. Another benefit could be the use of pipes to integrate the mass storage device in the internal file system of Android. The framework would then run in background and listen for events relating the piped directories and files. Every change to the piped structure would then be written back to the actual mass storage device. With this solution the mass storage device is mirrored into the Android file system.

Maybe with the next Android version Google adds native support for USB mass storage devices in their stock Android version while making this framework (nearly) obsolete. Some manufacturers already noticed that this feature is pretty useful especially when looking at the latest trend omitting a slot for micro SD-cards and relying only on the internal storage of a device. But we will see what Google brings next!

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/io/fsp/filesystemprovider.html

Appendix

A. Debugging applications via Wifi

When working with the USB features Android provides, an application on the device cannot be debugged as usual using a USB cable and plugging it into the computer. Android fortunately provides an easy solution for that. It allows debugging over Wifi just like using a USB cable.

To enable debugging over Wifi certain steps have to be done. The first step is to connect the device as usual to the computer and to execute the following command using adb:

Listing A.1: Restarting the device in Wifi debug mode

```
localhost:platform-tools mep$ ./adb tcpip 5555
restarting in TCP mode port: 5555
```

This command forces the device to restart the debugging functionalities in Wifi mode at port 5555. The device can now be used to debug over Wifi. To do this, the device's IP address has to be looked up in the Wifi settings[1]. With the IP address the connection can easily be established with another adb command:

Listing A.2: Connecting to the device over Wifi

```
localhost:platform-tools mep$ ./adb connect 192.168.2.108
connected to 192.168.2.108:5555
```

After that, deploying and debugging applications can be done, just as usual, in eclipse or the desired environment!

[1] The computer and the Android device, obviouly, have to be in the same (Wifi-)network.

B. Isochronous USB transfers

The introduction says that Android currently does not support isochronous USB transfers, and gives a reference to the official Android developer documentation[9]. But this does not seem to be appropriate for every device.

Some devices like the Samsung Galaxy S3 support audio output via a connected USB audio interface. Audio input seems to be unsupported. This feature is for example useful for USB headsets or docking stations which can play music. Unfortunately it is not part of the official Android. Nevertheless an application developer has no access to the isochronous transfers, they are hidden in the system and are only used to route the operating systems audio through the connected audio interface instead through the internal speaker.

The company beyerdynamic offers a headphone with an integrated amplifier and an USB audio interface. It is connected to the Android device over USB. The headphone can be used with some Android devices only, which support digital audio data via USB, for example the Samsung Galaxy S3 or S4, or the HTC Butterfly[3].

In fact, it seems that there are also people who get isochronous transfers to work on non rooted devices using native code and the system call *ioctl*[5, 13].

C. API difference

The section about the performance test stated that there is an API change between Android API level 18 and lower which affects the framework. In Android API level 18 the class UsbDeviceConnection offers two methods for performing bulk transfers[10]:

Listing C.1: Bulk transfers in UsbDeviceConnection

```
int bulkTransfer(UsbEndpoint endpoint, byte[] buffer, int offset, int
   length, int timeout);
int bulkTransfer(UsbEndpoint endpoint, byte[] buffer, int length, int
   timeout);
```

One method accepts an offset. It represents the index of the first byte where a read or write shall begin in the byte array. The framework developed in this thesis makes intense usage of the ability to specify the offset. The other method just begins reading or writing at offset zero in the byte array.

In API level lower 18 only the latter method is available. The only workaround which solves this issue is to create a temporary buffer, perform the bulk read or write and then copy the temporary buffer to the actual buffer at the desired offset. This yields to the overhead of an extra array copy. Listing C.2 shows how this is solved in the framework. More information can be found in the source code, especially in the class UsbMassStorageDevice.

Listing C.2: Workaround for the missing method in API level lower 18

```
@Override
public int bulkOutTransfer(byte[] buffer, int offset, int length) {
      if(offset == 0)
            return deviceConnection.bulkTransfer(outEndpoint, buffer,
               length, TRANSFER_TIMEOUT);

      byte[] tmpBuffer = new byte[length];
      System.arraycopy(buffer, offset, tmpBuffer, 0, length);
      int result = deviceConnection.bulkTransfer(outEndpoint,
         tmpBuffer, length, TRANSFER_TIMEOUT);
      return result;
}

@Override
public int bulkInTransfer(byte[] buffer, int offset, int length) {
      if(offset == 0)
            return deviceConnection.bulkTransfer(inEndpoint, buffer,
               length, TRANSFER_TIMEOUT);

      byte[] tmpBuffer = new byte[length];
      int result = deviceConnection.bulkTransfer(inEndpoint, tmpBuffer,
         length, TRANSFER_TIMEOUT);
      System.arraycopy(tmpBuffer, 0, buffer, offset, length);
      return result;
}
```

Bibliography

[1] androidcentral. Android Advanced: USB OTG on the Nexus 4. `http://www.androidcentral.com/android-advanced-usb-otg-nexus-4`, 2013. [Online; accessed March 21, 2014].

[2] Jan Axelson. *USB Mass Storage: Designing and Programming Devices and Embedded Hosts*. Lakeview Research LLC, 2006.

[3] beyerdynamic. beyerdynamic A 200 p- Mobile amplifier with digital-to-analog converter (DAC). `http://europe.beyerdynamic.com/shop/hah/headphones-and-headsets/at-home/headphones-amps/a-200-p-eu.html?SID=db7e0114b3661a365012c7d6d541691c&___store=en&___from_store=de`, 2014. [Online; accessed March 21, 2014].

[4] Microsoft Corporation. *Microsoft Extensible Firmware Initiative: FAT32 File System Specification*. Microsoft Corporation, 2000.

[5] eXtream Software Development. USB audio driver in USB Audio Player/Recorder PRO and Audio Evolution Mobile. `http://www.extreamsd.com/USBAudioRecorderPRO/`, 2014. [Online; accessed March 27, 2014].

[6] USB Implementers Forum. *Universal Serial Bus Mass Storage Class: Bulk-Only Transport*. USB Implementers Forum, 1999.

[7] Google Inc. USB Accessory — Android Developers. `http://developer.android.com/guide/topics/connectivity/usb/accessory.html`, 2011. [Online; accessed March 04, 2014].

[8] Google Inc. USB Host — Android Developers. `http://developer.android.com/guide/topics/connectivity/usb/host.html`, 2011. [Online; accessed February 28, 2014].

[9] Google Inc. UsbConstans — Android Developers. `http://developer.android.com/reference/android/hardware/usb/UsbConstants.html#USB_ENDPOINT_XFER_ISOC`, 2011. [Online; accessed March 04, 2014].

[10] Google Inc. UsbDeviceConnection — Android Developers. `http://developer.android.com/reference/android/hardware/usb/UsbDeviceConnection.html`, 2014. [Online; accessed March 21, 2014].

[11] Michael Opdenacker. Linux USB drivers. `http://free-electrons.com/doc/linux-usb.pdf`, 2009. [Online; accessed March 21, 2014].

[12] Seagate. *SCSI Commands Reference Manual*. Seagate Technology LLC, 2006.

[13] Various (stackoverflow). User mode USB isochronous transfer from device-to-host. http://stackoverflow.com/questions/7964315/user-mode-usb-isochronous-transfer-from-device-to-host, 2011. [Online; accessed March 27, 2014].

[14] Paul Stoffregen. Understanding FAT32 Filesystems. https://www.pjrc.com/tech/8051/ide/fat32.html, 2005. [Online; accessed March 21, 2014].

[15] USB-IF. Approved Class Specification Documents. http://www.usb.org/developers/devclass_docs#approved, 2014. [Online; accessed March 21, 2014].

[16] Wikipedia. File Allocation Table. http://en.wikipedia.org/wiki/File_Allocation_Table, 2014. [Online; accessed March 21, 2014].

[17] Wikipedia. GUID Partition Table. http://en.wikipedia.org/wiki/GUID_Partition_Table, 2014. [Online; accessed March 21, 2014].

[18] Wikipedia. Logical block addressing. http://en.wikipedia.org/wiki/Logical_block_addressing, 2014. [Online; accessed April 10, 2014].

[19] Wikipedia. SCSI. http://en.wikipedia.org/wiki/SCSI, 2014. [Online; accessed March 17, 2014].

[20] Wikipedia. USB. http://en.wikipedia.org/wiki/USB, 2014. [Online; accessed March 21, 2014].

[21] Wikipedia. USB On-The-Go. http://en.wikipedia.org/wiki/USB_On-The-Go, 2014. [Online; accessed March 21, 2014].